University of Malta

Department of Computer Science

Faculty of ICT

TATJA - A Test Automation Tool for Java Applets

Author: Matthew Xuereb

Supervisor: Dr. Ernest Cachia

Submitted in partial fulfillment of the requirements for the degree of B.Sc. I.T. (Hons.)

Abstract

Although there are some very good tools to test Web Applications, such tools neglect the facility to test Java Applets. With existing technologies, Java Applets can only be tested either by using Swing/AWT GUI Testing Tools or else by using tools that makes use of image recognition in order to capture and playback events. Such tools are not tailor made to test Java Applets and cannot be used to test Java Applets in their actual running environment, that is, in web browsers.

The implemented tool allows software testers to test Java Applets in their actual running environments. The tool is platform and browser independent and does make use of any third party scripting language. Tests cases are generated directly in Java.

To MV Maria Dolores

For being my inspiration throughout this course

Acknowledgments

I would like to thank my supervisor Dr. Ernest Cachia for the help, patience, and insightful advice throughout the course of this project.

A special thanks goes out to Mark Micallef, a friend and a mentor, for helping me to formulate the idea of this project and to Roberta Muscat for giving me the initial seed of inspiration to implement this testing tool.

Finally, I would like to thank my closest friends Lindsay and Daniel for their support and for helping to make these four years a truly memorable experience.

Table of Contents

1. Introduction 1
1.1 Background Theory 1
1.2 Software Testing Methods 2
1.3 Static vs. Dynamic Testing 3
1.4 Test Levels 4
1.5 Regression Testing 5
1.6 Automated Testing Tools 5
1.7 Summary 6

2. Testing Web Applications 7
2.1 Overview 7
2.1.1 Presentation Layer Testing 9
2.1.2 Business Layer Testing 10
2.1.3 Data Layer Testing 10
2.1.4 Web Application Test Automaton Requirements 11
2.1.5 Conclusions 13
2.2 Web Application Testing Tools 13
2.2.1 Content Testing Tools 14
2.2.2 Security Testing Tools 14
2.2.3 Functional Testing Tools 15
2.2.4 Conclusions 19
2.3 Summary 19

3. A Test Automation Tool for Java Applets 20
3.1 Java Applets Overview 20
3.2 TAJTA Overview 22
3.3 User action Capture and Generation 25
3.3.1 Using the Java Robot Class 26
3.3.2 Using an Applet Wrapper 26
3.3.3 Conclusion 27
3.4 Component Event and Method Invocation 28
3.5 Technologies to be used 29
3.5.1 XML 29
3.5.2 The Reflection API 30
3.6 Summary 31

4. Specification and Design 32
4.1 General System Architecture 32
4.1.1 Recording Mode 33
4.1.2 Playback Mode 35
4.2 TatjaML 37
4.2.1 Commands 38
4.2.2 Events 39
4.3 GUI Design 41
4.3.1 The Record Toolbar 41
4.3.2 The Playback Toolbar 42
4.4 Applet's Components Identification 43
4.5 TATJA 'Record' Design 44
4.5.1 'Record Instruction' Design 44
4.5.2 'Record GUI' Design 45
4.6 TATJA 'Playback' Design 47
4.7 Summary 48

5. Implementation 49
5.1 Implementation Overview 49
5.2 Package common 50
5.3 Package defenitionsdriver 50
5.3.1 Class DefenitionsDriver 51
5.4 Package dialogboxes 54
5.5 Package helpers 55
5.5.1 Class TestHelpers 56
5.6 Package tatja 60
5.6.1 Class AppletTester 61
5.6.2 Class RecorderHandler 62
5.6.3 Class TestPlayHandler 63
5.7 TATJA Configuration files 64
5.8 Summary 65

6. Testing and Evaluation 66
6.1 Testing Strategy 66
6.2 Test Case 1 – A Data Clustering Simulation Applet 67
6.2.1 Test Case Design 67
6.2.2 Test Case Execution 68
6.3 Test Case 2 – A Car Loan Calculator Applet 70
6.3.1 Test Case Design 71
6.3.2 Test Case Execution 72
6.4 Test Case 3 – A Frictionless Slope Simulation Applet 73
6.4.1 Test Case Design 74
6.4.2 Test Case Execution 75
6.5 Summary 77

7. Conclusions and Future Work 78
7.1 Summery of Results 78
7.2 Future Work and Research 78

7.3 Conclusion 79

References 81

Bibliography 85

A. User Manual 86
A.1 Requirements 86
A.2 Installation and Execution Instructions 86
A.3 TATJA in Record Mode 87
A.3.1 The Record Toolbar 87
A.3.2 Loading the Applet to be tested 88
A.3.2 Loading/Creating TATJA Test Classes 89
A.3.3 Selecting/Creating TATJA Test Methods 90
A.3.4 Test Case Recording 91
A.3 TATJA in Playback Mode 95
A.3.1 The Playback Toolbar 95
A.3.2 Loading Applet and Test Class 96
A.3.2 Executing a Test Case 96
A.4 TATJA Test Suite 96
A.4.1 Creating a Test Suite 96
A.4.2 Editing/Programming a Test Suite 97
A.4.3 Executing a Test Suite 98

B. TatjaML Sample Document 100

C. Contents of the CD ROM 105

D. Permissions and Clarifications 106
D.1 Permission to use working Applets 106
D.1.1 Permission email from Mr. Damien Martin 106
D.1.2 Permission email from Profs. Deepak Kumar 107
D.2 The TATJA Logo 108

List of Figures

Figure 1: Use Case Diagram 22
Figure 2: Recording process Activity Diagram 24
Figure 3: Playback process Activity Diagram 24
Figure 4: An XML markup for a news article 29
Figure 5: General System Architecture 32
Figure 6: TATJA in recording mode 34
Figure 7: TATJA in playback mode (When Used Manually) 35
Figure 8: TATJA in playback mode (When invoked programmatically) 36
Figure 9: An extract from a TatjaML document 37
Figure 10: An example of a TatjaML command 38
Figure 11: An example of a TatjaML event 39
Figure 12: Design of the user interface 41
Figure 13: Design of the toolbar in playback mode 42
Figure 14: Example of the dot notation as adopted by TATJA 44
Figure 15: Example of a TatjaML definition for a JRadioButton component 46
Figure 16: A sketch of the ‘record’ dialog box 47
Figure 17: A screen shot of TATJA in Sun’s Applet Viewer 61
Figure 18: A screen shot of TATJA with the mouse pointer on a component and with the Record Dialog Box visible 63
Figure 19: A screen shot 67
Figure 20: A screen shot 69
Figure 21: A screen shot 70
Figure 22: A screen shot 71
Figure 23: A screen shot 72
Figure 24: A screen shot 73
Figure 25: A screen shot 74
Figure 26: A screen shot 75
Figure 27: A screen shot 77
Figure 28: Screen shot of TATJA Record toolbar 82
Figure 29: Screen shot of the Load Applet Dialog Box 83
Figure 30: Screen Shot of the Test Class Chooser Dialog Box 85
Figure 31: Screen shot of the Test Method Chooser Dialog Box 86
Figure 32: A screen shot 87

Figure 33: A screen shot 88
Figure 34: A screen shot 88
Figure 35: A screen shot 89
Figure 36: A screen shot 90
Figure 37: A screen shot 90
Figure 28: A screen shot 92

Chapter 1
Introduction

1.1 Background Theory

Software testing is the process used to measure the quality of computer software in terms of the defects found. In the book entitled *The Art of Software Testing* [3], the author argues that software testing is not the process of showing that the program does not contains errors, but it is the other way round, that is, the process of executing a program with the target of finding errors.

There are different methods and approaches to software testing. However before applying such methods and approaches, a software test engineer should understand the basic principles that guide software testing. Pressman [1] identifies a set of testing principles that are adapted from the book of Davis [31], *201 Principles of Software Development*. These basic principles are listed and explained below:

- **Testing should begin 'in the small' and progress towards testing 'in the large'.** The first tests planned should focus in individual components and then shifts focus to find errors in the entire system.
- **Exhaustive testing is not possible.** It is impossible to execute every combination of paths during testing. It is possible however, to adequately cover program logic and to ensure that all conditions have been met.

- **All tests should be traceable to customer requirements.** The objective of software testing is to uncover errors. The most severe defects are those that cause the program to fail to meet its requirements.
- **Tests should be planned long before testing begins.** Detailed definition of test cases can begin as soon as the design model is ready.

1.2 Software Testing Methods

Software testing methods are traditionally divided into *black box testing* and *white box testing*. These approaches are used to describe the point of view that a software test engineer takes when designing test cases.

Black box testing is concerned about the functionality of the program rather than about the internal behaviour or structure of the system. It is called *black box* for the simple reason that no knowledge of the workings of the program is used as part of the testing. Only inputs and outputs are considered since the program is thought of as being invisible within a black box [2]. Black box testing usually requires through test cases to be provided to the tester who then can simply verify that for a given input, the output value or behaviour is the same as the expected value specified in the test case. Black box testing attempts to find errors in the following categories [1]:

- Incorrect or missing functions
- Errors in data structures or external database access
- Behaviour or performance errors
- Initialisation and termination errors

White box testing (sometimes also called *Glass Box Testing*) makes use of the knowledge of the internal data structures, code and algorithms of the system. Using white box testing methods, the test engineer can derive test cases that [1]:

- Guarantee that al independent paths within a module have been exercised at least once
- Exercise all logical decisions on their true and false sides
- Execute all loops at their boundaries and within their operational bounds
- Exercise internal data structure to assure their validity

1.3 Static vs. Dynamic Testing

There are two basic ways of how to test a piece of software – either static or dynamic. Static testing is when a program is tested without running it. Static testing is primarily syntax checking of the code and manually reading of the code or documents to find errors. During static testing, test engineers perform formal reviews, checks for correct coding standards, and also perform code walkthroughs. Static testing can be done before dynamic test execution.

On the other hand, dynamic testing is about testing the dynamic behaviour of the code, i.e. testing a running program. In dynamic testing the program must actually be compiled and executed. Dynamic testing involves working with the software, giving input values and checking if the output is as expected.

1.4 Test Levels

There are four main levels of software testing (As agreed by many software testing books). These are:

1. Component Testing
2. Integration Testing
3. System Testing
4. Acceptance Testing

Component testing (also called *Unit Testing*) is the process of testing an individual portion of code (that is, a component) to determine whether it functions properly [4]. Component testing is usually performed by the software developer and not by the software test engineer. If component testing is done properly, later testing will be more successful.

Integration testing is the phase in which individual software modules are combined and tested as a group. The goal of integration testing is to expose faults in the interfaces and in interactions between integrated components.

System testing is carried out on the complete integrated system in order to evaluate the system's compliance with its specified requirements. Its goal is to verify that the system meets the functional specification and all the non-functional requirements.

Acceptance testing is formal testing with respect to the user needs, requirements and business process conducted to determine if the system satisfies acceptance criteria and to decide whether or not to accept the system. Acceptance testing is conducted either by the end-user, or the customer, or the client.

1.5 Regression Testing

Regression testing is the repeated testing of an already tested program after it was modified. The aim behind regression testing is to discover any defects introduced or uncovered as a result of the changes. Such testing is performed when the software or its environment has changed. Regression testing may be performed at all test level, and applies to functional, non-functional and structural testing. Regression test suites are executed many times. Due to this reason, regression testing is a strong candidate for automation testing.

1.6 Automated Testing Tools

Test automation is the use of software to control the execution of tests. Automated testing tools can enhance the testing effort provided that expectations and managed tool issues are understood, and a tool compatible with the system-engineering environment is selected [4]. It is very important that the tool matches the task at hand, and that the tests lend themselves to automation.

Sometimes it is determined that no tool available on the market completely suites the project's needs. In such case, it is necessary to decide whether to develop a unique solution in the form of scripts or custom tools, or to rely solely on manual testing. In other cases, there may be commercially available tools that would work for the testing effort, but offer more features than they are needed [4].

Many software test automation tools provide record and playback features that allow users to record interactively user actions and replay it back any number of times comparing actual results to those expected. Test automation is expensive and it is an addition, not a replacement to manual testing.

1.7 Summary

In this introductory chapter, an overview of the theory behind software testing is given. The different software testing methods, static testing and dynamic testing are explained. The different test levels and regression testing are discussed. At the end of this chapter and overview of automated testing tools is given.

Chapter 2
Testing Web Applications

The goal of testing Internet-based application is the same as that of testing traditional applications. The main aim is to uncover errors in the application before deploying it to the internet.

2.1 Overview

A Web-based application has many failure points that should be considered during testing. Myers [3] identifies a list of some of the main challenges associated with testing such applications. These are:

- **Large and varied user base.** Internet users possess different skill sets, employ a variety of browsers, and use different operating systems or devices. Users access the internet using a wide range of connection speeds.

- **Business environment.** If operating an e-commerce site, issues such as tax calculation, shipping costs, tracking customer profiles, and more, should be considered.

- **Locales.** Users may reside in other countries. In such case internationalisation issues such as language translation, time zone, and currency conversion, should be considered.

- **Testing environments.** To properly test a web application, it is required to duplicate the production environment. This means that Web Servers, Application Servers, and Database Servers that are identical to the production equipment should be used. For the most accurate testing results, the network infrastructure will have to be duplicated as well. This includes routers, switches, and firewalls.

- **Security.** Since the website is open to the world, it should be protected form hackers. Hackers can bring the website to a grinding halt with denial-of-service attacks or else rip off the customers' credit card information.

Another major testing challenge is testing browser compatibility. Today there are several different web browsers on the market, and each behaves differently. Although standards exist for browser operation, most vendors enhance their browsers to try and attract a loyal user base [3]. Unfortunately, this causes the browsers to operate in a non-standard way.

Although many challenges exist when testing Web-based applications, one should narrow the testing efforts to specific areas. Myers [3] identifies some of the most important testing areas of a Web-based application. The latter divides a Web-based application into three tiers:

1. **Presentation Layer.** The layer of an Internet application that provides the graphical user interface.
2. **Business Logic Layer.** The layer that models the business processes such as user authentication and transactions.
3. **Data Access Layer.** The layer that houses data used by the application or that is collected from the end user.

2.1.1 Presentation Layer Testing

Testing the presentation layer consists of finding errors in the graphical user interface of the web application. This important layer provides the curb appeal of the web site, so detecting and correcting errors in this layer are critical to presenting a quality, robust website.

Presentation layer testing is very work intensive. However the work can be segmented into three major areas [3]:

1. **Content testing.** The test engineer should search for errors in the font type, screen layout, colours, graphics resolutions, and other features that directly affect the end-user experience. In addition, the tester should check the accuracy of the information on the website.

2. **Website architecture.** The website architecture should be tested to find navigational and structural errors. The tester should search for broken links, missing pages, wrong files, or anything that sends the user to the wrong area of the site.

3. **User environment.** The test engineer should test for different web browsers and web browser's versions and operating system configurations. User environment testing becomes more complex when the web-application makes use of client-side scripting and/or Applets since every browser has a different scripting engine or virtual machine to run scripts and code on the client's machine.

2.1.2 Business Layer Testing

Business layer testing focuses on finding errors in the business logic of the web application. There are certain areas of the business layer that should always be tested. These areas include [3]:

1. **Performance.** Test to see whether the application meets the documented performance specifications. These are generally specified in response times and throughput rates.
2. **Data Validity.** Test to detect errors in data collected from customers.
3. **Transactions.** Test to uncover errors in transaction processing, which may include items such as credit card processing, emailing verifications, and calculation sales tax.

2.1.3 Data Layer Testing

Testing of the data layer consists of testing the database management system that the web-application uses to store and retrieve information. As with the other tiers, there are certain areas that should be taken into consideration when testing the data layer. These are [3]:

1. **Response Time.** Slow web-applications cause unhappy users. Therefore, it is important that the test engineer makes sure that the web-application responds in a timely manner to user requests and actions. *Response Time* testing in this layer does not include timing page loads, but focuses on identifying database operations that do not meet performance objectives. When testing this layer, the tester should ensure that individual database operations occur quickly so as not to bottleneck other operations.

2. **Data Integrity.** *Data Integrity* testing is the process of finding inaccurate data in your database tables. This test is different

from data validation. Data validation testing tries to find errors in data collection. Data integrity testing strives to find errors in how data is stored. There are many factors that may affect hot the data is stored. For example the data type and length can cause data truncation or loss of precision. There are also time zone issues. For instance should time be stored based on the location of the client, the web server, the application server, or the RDBMS? The tester should also check for the accuracy of the lookup/reference tables (such as tax, time zone information, etc…) used by the web-application.

3. **Fault tolerance and recoverability.** If the web-application relies on an RDBMS, then the system must stay up and running. There is very little, if any, time available for downtime in this scenario. Thus, the test engineer should test the fault tolerance and recoverability of the database system. One goal of database operations, in general, is to maximise MTBF and minimise MTTR. Note that these values should be specified in the requirements documentation of the web-application. The tester's goal when testing the database system robustness is to try to exceed these numbers.

2.1.4 Web Application Test Automaton Requirements

In order to assure the quality of a web application it is required that a number of test cases are designed and executed automatically and repeatedly to demonstrate the required web application's capabilities and behaviors under different circumstances.

Miller [29] identifies the basic elements that are required by web application testing tools in order to achieve this. According to Miller,

typical elements of tests should involve the characteristics explained in the following list:

- **Browser Independent.** Tests should be realistic, but not dependent on a particular browser, whose biases and characteristics might mask a website's problems.

- **No Buffering, Caching.** Local buffering and caching – often a way to improve apparent performance – should be disabled so that times experiments are a true measure of the *Browser-Web-Website-Web-Browser* response time.

- **Fonts and Preferences.** Most browsers support a wide range of fonts and presentation preferences, and these should not affect how quality on a website is assessed or assured.

- **Object Mode.** Edit fields, push buttons, radio buttons, check boxes, etc... All should be treatable in object mode, that is, independent of the fonts and preferences.

  Object mode operation is essential to protect an investment in test suites and to assure that test suites continue operating when website pages experience change. In other words, when buttons and form entries change location on the screen, the tests should still work.

- **Tables and Forms.** Even when the layout of a table or form varies in the browser's view, test of it should continue independent of these factors.

- **Frames.** Windows with multiple frames should be processed as if they were multiple single-page frames.

2.1.5 Conclusions

It is concluded that there are basically three main areas of testing as regards to web applications. These are:

1. **Content Testing.** Testing of the visible part of the web site, including testing to verify that the web pages are rendered the same on the various available web browsers.

2. **Security Testing.** Testing for security issues within the web application.

3. **Functional Testing.** Testing for the functionality (Business Logic) of the application.

Such tests can be automated. In the next section of this chapter some testing tools that are specifically designed to test web applications are and some if the main web application testing guidelines are discussed.

2.2 Web Application Testing Tools

There is a large number of Web Application testing tools. The *Software QA/Test Resource Centre* [5] identifies more than 370 of such tools. Different tools can be used for different purposes and for different technologies.

Some of these tools are discussed in this section. The discussed tools are categorised in three categories as discussed in *Section 2.1.5* of this chapter.

2.2.1 Content Testing Tools

The W3C Consortium [30] has devised a number of guidelines and free online tools as regards to the content of web pages. Such tools include:

- **Markup Validator.** This tool is used to check for the markup validity of web documents written in HTML, XHTML, SMIL, and more [32]. This tool is also used to check if the web application is 'accessible to a wider range of people with disabilities, including blindness and low vision, deafness and hearing loss, learning disabilities, cognitive limitations, limited movement, speech difficulties, photosensitivity and combinations of these' [32]. This tool can be accessed online through the following URL: http://validator.w3.org/

- **Link Checker.** This tool is used to check hyperlinks. It is very useful to find dangling links and more [33]. This tool can be accessed online from: http://validator.w3.org/checklink

- **CSS Validator.** This tool is used to check and validate CSS definitions [34]. It can be accessed online from: http://jigsaw.w3.org/css-validator/

2.2.2 Security Testing Tools

With the popularity of the internet and web applications always increasing, attackers and malicious web users are also enhancing their techniques to attack such applications. It is therefore important for web application developers test engineers to keep up-to-date with the latest security tools and products that can help in keeping a web application secure form such vulnerability attacks.

There are a lot of commercial tools that can be used check web applications against security threats. The pioneers in this area are *Acunetix* [35], *SecurityInnovation* [36] and *SpyDynamics* [37]. Such companies provides various static and dynamic tools that checks web applications for threats such as *Injection Flaws*, *Cross Site Scripting*, *Insecure Communications* and more.

Due to the importance of security testing in web applications, a non-profit organisation named *OWASP* (Open Web Application Security Project) [38] has been set up in order to perform research and keep up-to-date with the web security issues.

The OWASP organisation is most renowned for the *OWASP Top Ten Project* [39]. This is a guideline for developers and testers highlighting the top ten malicious web applications attacks and how to prevent them. This guideline also suggests the different tools to use in order to test for such attacks.

2.2.3 Functional Testing Tools

There are loads of web application testing tools that are used to test the functionality and the business logic of the application. A number of these tools provide capture/playback whilst others can only be used programmatically. Some of these tools are explained below.

- **Cactus.** Cactus is a simple framework for unit testing server-side java code such as *Servlets*, *EJBs*, *Tag Libs*, and more. This tool is an extension of the Java API *JUnit* [6]. Cactus is an open source *Jakarta* project and it can be downloaded free of charge from http://jakarta.apache.org/cactus/.

- **FormFlood.** This is a utility program that *will put a lot of plausible data into website forms* [7]. FormFlood is a tool to test web forms to make sure they can handle a load, that all of the links work, and that all of the forms submit correctly [7]. This is another open source project and it can be downloaded from http://formflood.sourceforge.net/.

- **HTMLUnit.** *HTMLUnit* is a 'browser for Java Programs' [8]. It models HTML documents and provides an API that allows the test engineer to invoke pages, fill out HTML forms, click links and more. HTMLUnit is not a generic testing framework. It is specifically a way to simulate a browser for testing purposes and is intended to be used with another testing framework such as *JUnit* [8]. This is an open source project and it can be downloaded from http://htmlunit.sourceforge.net/.

- **HTTPUnit.** HTTPUnit is a testing tool written in Java that emulates the relevant portions of browser behaviour, including form submission, JavaScript, basic http authentication, cookies and automatic page redirection, and allows java test code to examine returned pages [9]. Preferably this tool should be used in conjunction with *JUnit*. This is a free tool and can be downloaded from http://www.httpunit.org/.

- **Apache JMeter.** This tool may be used to test performance on both static and dynamic web applications. Apache JMeter can be used to simulate a heavy load on a server, network or object to test its strength or to analyse overall performance under different load types [10]. This is another open source Jakarta project that is available from http://jakarta.apache.org/jmeter/.

- **MaxQ. '**MaxQ is a web functional testing tool' [11]. It records the actions done be the user and turns them into a Python script.

The script can then be play backed [11]. This is a free tool that can be downloaded from http://maxq.tigris.org/.

- **WAPT.** WAPT is a load and stress testing tool that provides an easy-to-use way of testing web sites, web servers, and intranet applications with web interfaces [12]. This is a commercial tool designed for Microsoft Windows and it can be purchased from http://www.loadtestingtool.com/.

- **Selenium.** Selenium is a software testing framework in which tests can be written as HTML tables or coded in a number of popular programming languages and can be run directly in most modern web browsers. Selenium can be used on Windows, Linux, and Macintosh [13]. This is on open source tool that can be downloaded from http://selenium.openqa.org/.

- **iMacros.** iMacros is a testing tool that can be used to perform in-browser functional, performance, and regression testing of web applications. This is a commercial tool and it can be downloaded from http://www.iopus.com/imacros/web-testing.htm. iMacros claims that is 'the only solution that can test Flash, Flex, Silverlight, ActiveX and Java Applets inside the browser' [14]. iMacros makes use of image recognition to identify different media components (such as buttons, links, form fields, etc...). Note that this tool is platform dependent since it works only on Microsoft Windows.

Although all of the above functional testing tools have their own pros and cons, in my opinion *Selenium* is the best tool to use for the testing of the business logic of web applications. The following more in-depth description of Selenium explains why I think that Selenium is the best tool to use.

Selenium executes the tests directly in an actual web browser. This tool is platform independent since it is compatible with Internet Explorer, Mozilla and Firefox on Windows, Linux and Macintosh, and it is also compatible with safari on the Mac.

Selenium has two modes of operation:

1. **Core.** *Selenium Core* is written in JavaScript/DHTML. These technologies are used to embed a test automation engine in the browser. Note that the web browser should be JavaScript enabled for *Selenium Core* to be effective.

2. **Remote Control (RC).** *Selenium Remote Control* is a tool that allows a tester to write automated web application user interface tests in any programming language against any HTTP website using any JavaScript enabled web browser. *Selenium RC* comes in two parts:
   I. A server which can automatically launch and kill supported browsers, and act as an HTTP proxy for web requests from these browsers.

   II. Client libraries for the testers' preferred programming language.

Apart from *Selenium Core* and *Selenium RC*, there is also a Firefox extension for Selenium called *Selenium IDE*. This plug-in allows a tester to record, edit, and deploy tests. By using this plug-in, the tester can choose to either use its recording capability or else the tester may edit the test scripts by hand. (Note: All information is taken from [13]).

2.2.4 Conclusions

Although as I said Selenium is in my opinion the best web-application testing tool to use, it still neglects an important feature. Selenium is not capable of testing Java Applets.

According to my research, the only web-application testing tool that can be used to test Java Applets in their actual working environment is the *iMacros* [14] tool that is developed by *iOpus*™ [15]. However, unlike Selenium, this tool is not distributed free of charge. Another draw back is that *iMacros* is a platform dependent tool since it can only be used with Internet Explorer and Firefox on Microsoft Windows only.

2.3 Summary

In this chapter, the testing of web applications was explained in some detail. Different kinds of testing tools were analysed and criticised. From this chapter, it is concluded that although there are many tools that are used to test web applications, there isn't a tool that is tailor made specifically to test Java Applets. With the current tools Java Applets can only be tested either by using Swing testing tools (Tools that treats a Java Applet as a Java Application) or else by using tools that makes use of image processing.

Chapter 3

A Test Automation Tool for Java Applets

As described in the previous chapter, although there are a lot of tools that can be used to test Web Applications, there is not a tool available that can be used to test Java Applets in web browsers and that is browser and platform independent.

The aim of this dissertation is to develop a browser and platform independent tool that is able to test Java Applets in their actual environments. We are going to call this tool ***TATJA***, short for ***A Test Automation Tool for Java Applets***. Naturally, this tool should be developed using Java as implementation language. The reasons for this being that:

1. Java is a platform independent language, and
2. Since Applets are implemented using Java, it is much easier to access the components of the Applet with the same language that implements them.

3.1 Java Applets Overview

An Applet is a software component that runs in the context of another program. A Java Applet is an Applet that is written in the Java programming language in order to be executed in a web browser. Applets can be included in an HTML page much in the same way that an image is included in a page.

Applets are used to provide interactive features to web applications that cannot be provided by HTML. Although with the introduction of AJAX the popularity of Applets has decreased, Java Applets are still being used in critical parts of web applications, such as FTP clients, download/upload managers, and more.

Since Java Applets are delivered in the form of Java byte code, Java Applets can be executed by browsers on different platforms including Windows, Linux, Mac OS, UNIX and more. The byte code of the Applet is downloaded from a web server. The browser either embeds the Applet into a web page or else opens a new window or tab showing the Applet's user interface. Due to the fact that Java Applets can have malicious code, many web browsers execute them in a sandbox. A sandbox is a security mechanism for safely running programs.

During the development process, Java Applets are tested using Sun's *AppletViewer*. This is a stand alone tool for testing Java Applets. Although the Applet may be working correctly on the *AppletViewer*, many times a number of bugs and defects are visible when the Applet is executed in its actual environment, i.e. in web browsers.

Despite the popularity of Java Applets, there are not any tools specialised for testing Java Applets in web browsers. Java Applets can either be tested using tools that are used to test *Java Swing Applications* or else by using generic tools that can also be used to test other type of media files such as *Flash*, *Shockwave* and *ActiveX*. The disadvantage of using the former is that in such testing tools, a Java Applet is treated as a conventional application rather than as an Applet. If using the latter, although Applets are tested in web browsers, such tools are browser specific and since they are not Java specific, they makes use of image processing in order to identify different media components. Such tools also make use of third party scripting

languages, therefore in order to use them the software test engineer has to learn new syntax.

3.2 TAJTA Overview

Figure 1 illustrates a *Use Case Diagram* of the proposed system. TATJA should be a record/playback testing tool. As discussed in Chapter 2, such tools allows a Test Engineer to first record Test Cases by mimicking the user actions, and then the recorded test cases can be play backed for a number of times to test the system.

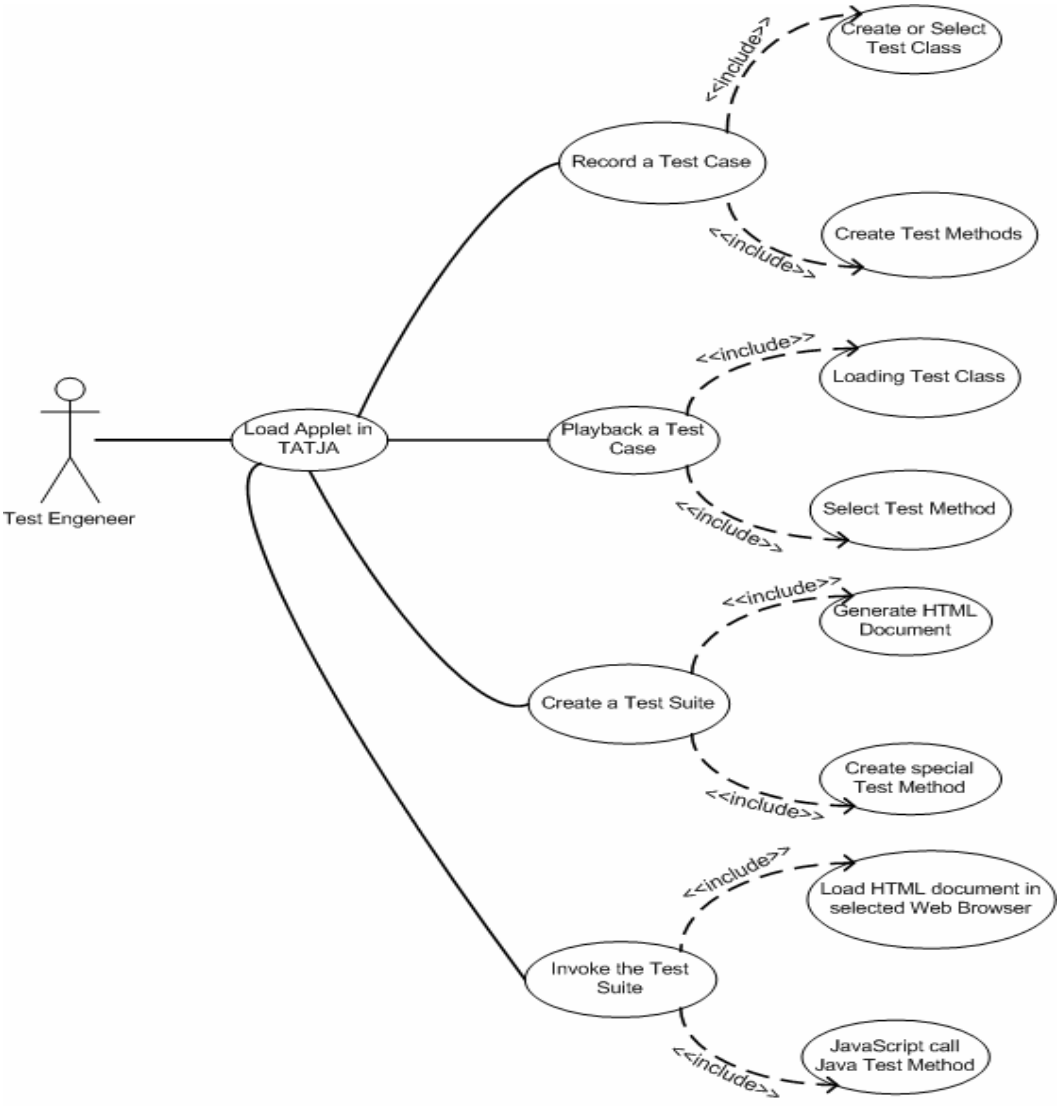

Figure 1: Use Case Diagram

The proposed tool should be able to be executed on any web browser and on any operating system. As it is clearly illustrated in Figure 1, the test engineer should first load the Applet to be tested in the tool. The proposed system is going to be operated in two modes:

1. Recording Mode
2. Playback Mode

During the recording mode, the user of the tool should be able to record test cases. The test cases generated by TATJA should be written using the Java Language. By using Java as the language for the test cases (Instead of using another third party scripting language), the test engineers are going to be able to change, edit, or enhance the test cases without the need of learning new syntax.

As shown in Figure 1, the recording process consists of first loading the Applet to be tested within TATJA. Then the user should create (Through the use of the tool) or make use of a test class in order to *code* the recorded test cases. In TATJA it is going to be assumed that one test case is equivalent to one Java method within a particular test class. The user can create a number of different test methods and through the use of the tool, the user should be able to record these test cases.

Since the recorded test cases are going to be written completely in Java, the test engineer can perform changes or enhance the test cases and can also make used of existent Java classes, libraries and helpers. Figure 2 illustrates an activity diagram of the recording process.

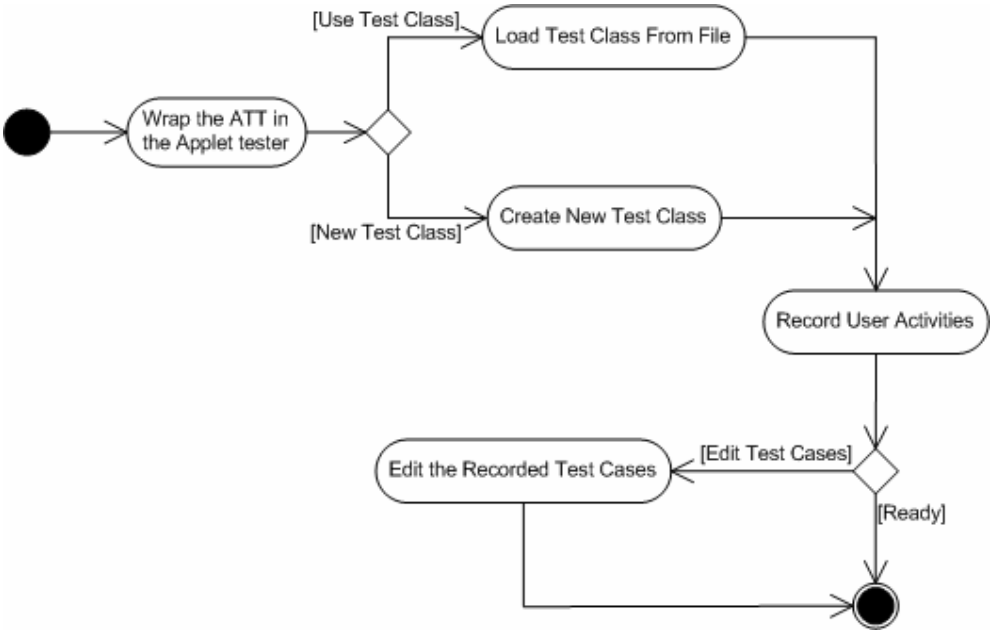

Figure 2: Recording process Activity Diagram

During the playback mode, the recorded test cases are play backed so that the test engineer can analyse the performance of the Applet. As illustrated in Figure 1, the playback process consists of first loading the Applet to be tested in the tool. The user should be able to load a particular test class that has in it the desired recorded test cases. The user can then playback the test cases in order to obtain the results. Figure 3 illustrates an activity diagram of the playback process.

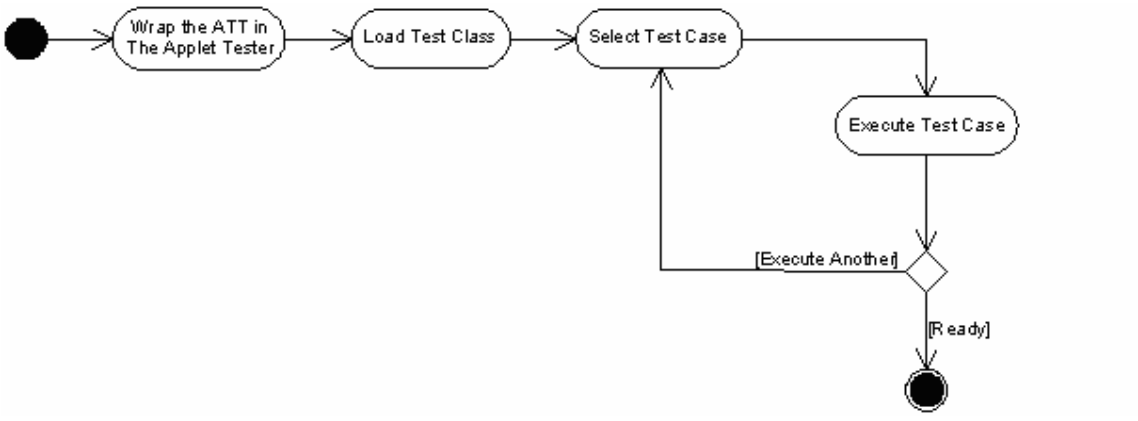

Figure 3: Playback process Activity Diagram

Note that the playback process explained above requires the user to load the Java Applet to be tested in the Applet testing tool. Since in the majority of cases TATJA is going to be used together with other web-application testing tools such as *Selenium*, TATJA should have the capability of executing the TATJA test cases without the need of manually loading the Applet testing tool.

The solution for this problem is to allow the user to create a test suite. A test suite is one single Java method that contains code that invokes a number of recorded test cases. As it is clearly shown in Figure 1, a user should be able to create test suite. Creating a test suite includes the creation of an HTML document in order to embed the Applet to be tested within the generated Test Suite. In order to execute this Test Suite, the user has to programmatically (Through the use of a tailor made API) loads the generated HTML document in any web browser and the test cases are play backed automatically.

3.3 User action Capture and Generation

At this stage of the project the functionality of the project is defined. The next step is to decide about how to implement it and the techniques to use. Basically there are two ways of how to capture and then generate user actions of different Applet's components in a web browser. There two ways are:

1. Making use of the *Java Robot Class* in order to control the mouse and keyboard actions.
2. Wrapping the Applet to be tested within another Applet and programmatically generate the required mouse and keyboard events.

This subsection discusses and compares the above two options and concludes which option is the most likely to be the best solution.

3.3.1 Using the Java Robot Class

The *Java Robot Class* is a Java native class that is used mainly to generate native system input events for the purpose of test automation, self running demos, and other applications where control of mouse and keyboard are needed. Note that using the Robot Class is different from posting events to the AWT event queue or Swing components because the events are generated in the platform's native input queue. [27]

In order to male use of the Robot class, absolute screen positions of the Applet's components are required since many of the methods of the Robot class are of a type where they requires parameters such as the screen coordinates and they place the mouse pointer in that defined position. Although it might at first seem that by using this class things are going to be easy and straight forward, it is not the case. Since many of the methods of the Robot class require the absolute screen positions, it is required to develop a mechanism that can convert relative component screen positions into absolute screen positions.

It is also required to develop a way to uniquely identify each Applet's component and get information such as their relative position to the Applet. Although by using the Robot class events can be generated on each component of the Applet, native Java method calls of the components, such as *getText* and *setText* cannot be invoked.

3.3.2 Using an Applet Wrapper

Software wrapping is a technique that wraps a piece of executable software within newly developed software in order not to rewrite the old piece of executable software from scratch. This technique is used mainly with the replacement of old legacy system. With wrapping old code is left in its native environment and is connected to the new software by an API or a CORBA type interface [28]. This method is

used mainly to gain time, however sooner or later 'the legacy functions must be reimplemented within the context of the new environment' [28].

Software wrapping can potentially be the solution to this project. In order to be able to execute the tool in a web browser, the tool can be itself an Applet. The Applet to be tested is wrapped within the Applet tool and thus can then gain access to the components of the Applet to be tested. Since the Applet to be tested is wrapped within the Applet testing tool (TATJA), each method that is executed on the tool is also indirectly being executed in the Applet to be tested.

If the wrapping approach is used, then the events can be generated programmatically and since the tool can gain access to each component of the Applet to be tested, then native Java methods of the components can also be invoked. Using this approach, a mechanism to uniquely identify each Applet's component and get relevant information about it still has to be devised.

3.3.3 Conclusion

It is concluded that the best technique to use for the implementation of this project is that of the wrapping method. The main reason for this being that once the Applet to be tested is wrapped within the Applet testing tool, the tool can gain total control over the entire Applet to be tested components. The extra mechanism regarding the issue of uniquely identifying each Applet's component that has to be implemented, would still have to be implemented having chosen the other technique, that of the Robot class.

3.4 Component Event and Method Invocation

In order to be able to record and then playback a particular user action, TATJA has to first acquire the privileges to invoke a method or event on the components of the Applet that is being tested. There are many actions that can be done on each Applet component. For instance on a radio button one can click, double click, set as checked, set as unchecked, and more.

Since there are infinitely many actions and events that can be done on each Applet (Swing) component, it is impossible to programmatically hard code each and every possible action in the coding of the Applet testing tool. In certain cases, some developers develop new components that extend from the Swing API. If the methods and events that can be performed by TATJA are hard coded, then the methods and events of such components cannot be invoked by the tool.

Due to the reasons explained above, the tool should have a mechanism that lets the user to define the commands (i.e. methods and events) that can be invoked on each Applet component. The most effective way of implementing this is to have an XML file with the different Applet components together with their respective commands (i.e. methods and events) that can be used by TATJA. When recording a particular test case, TATJA should only allow the user to perform the actions that are listed as commands of that particular component in the said XML file.

In the playback process, the tool should be able to dynamically invoke the events or methods that are defined in the XML file for each component.

3.5 Technologies to be used

TATJA is implemented in Java. The two main technologies used in this project apart from the Java Standard Edition (J2SE) are *XML* and the *Java Reflection API*.

3.5.1 XML

Extensible Markup Language (XML) is a markup language for documents containing structured information. XML makes use of tags. This markup language has been a W3C [16] recommendation since February 1998. *Figure 4* shows an example taken from [23] of an XML markup for a news Article.

```
<article>
      <title>Simple XML</title>
      <date>July 15, 2003</date>
      <author>
            <firstname>Carpenter</firstname>
            <lastname>Cal</lastname>
      </author>
      <summery>XML is pretty easy.</summery>
      <content>
      Once you have mastered XHTML, XML is easily
      learned. You must remember that XML is not for
      displaying information but for managing
      information.
      </content>
</article>
```

Figure 4: An XML markup for a news article

As shown in *Figure 4*, XML documents contain only data, not formatting instructions, so applications that process XML documents must decide how to manipulate or display the document's data. For example a Personal Digital Assistant (PDA) may render an XML document differently from a mobile phone or a desktop computer.

XML permits document authors to create markup for virtually any type of information. This extensibility enables document authors to create

markup language for describing data such as mathematical formulas, music, news, and more. Examples of XML markup languages include:

- **MathML**. XML for describing mathematical notations. [17]
- **GraphML**. XML for describing graphs. [18]
- **MusicXML**. XML for describing music notation. [19]

XML is going to be used in this project in order to define a markup language for TAJTA. This markup language is going to be called **TatjaML** and it is going to be used to define the commands (events and methods) that the user of the tool can invoke on each different component.

3.5.2 The Reflection API

In computer science, reflection is the process by which a computer program can 'observe and modify its own structure and behaviour' [20]. Reflection is commonly used by programs which require the ability to examine or modify the runtime behaviour of applications. [21].

> 'The use of reflection is an important technique for improving productivity. Reflection facilitates development of programs that are easily adapted to requirement changes. With reflection one can develop software engineering tools that examine [and/] or produce code. Reflection facilitates testing and problem determination by facilitating the automation of more tedious tasks. In general, reflection improves the flexibility, extensibility, and reusability of one's code.' [22]

The Java programming language (from version 1.4 upwards) contains a highly effective reflection facility. A description of the classes and methods of the Java Reflection API – *java.lang.reflect* can be found on:

http://java.sun.com/j2se/1.4.2/docs/api/java/lang/reflect/package-summary.html

The reflection API is going to be used in this project in order to be able to access the different components that make up the Applet. Reflection is also going to be use in order to dynamically perform the different commands on the different components that are defined in the *TatjaML* XML file.

3.6 Summary

In this chapter an overview of the testing tool is given. The name of the tool, TATJA, is also defined. The way to tackle the problem of how to capture and then generate the user action is discussed. A short description of the technologies to be used is given together with a description of where and why these technologies are used. The requirements of the tool are defined. These requirements are listed below:

The tool should:

- Be able to test Java Applets in web browsers
- Be browser and platform independent
- Have test capture (record) and playback capabilities
- Have a mechanism to program and execute regression tests
- Make use of the Java language to program the recorded test cases
- Be flexible and allow the user to define his/her own possible component actions (by means of an XML definition file)

Chapter 4

Specification and Design

4.1 General System Architecture

As it is explained in the previous chapter, it is decided that TATJA should be an Applet that wraps within it the Applet to be tested. Once that the Applet to be tested is wrapped inside the Applet testing tool, the testing tool can gain access of the components, then it can be used to record and playback user actions.

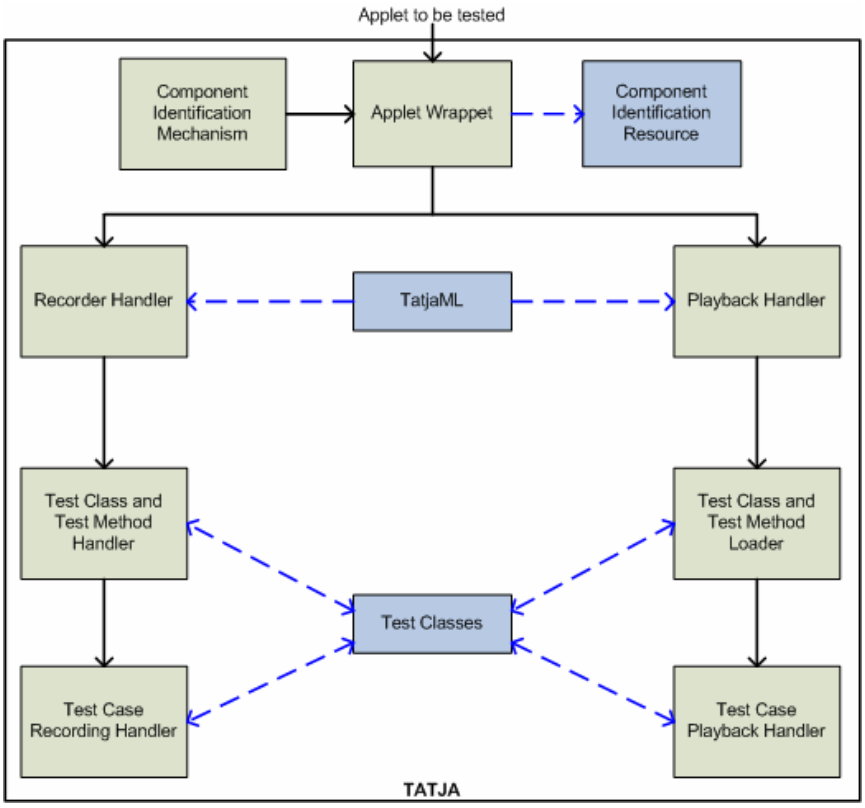

Figure 5: General System Architecture

Figure 5 illustrates a diagram of the general system architecture of the proposed system. As it is clearly explained in Figure 5, the Applet to be tested is first loaded and wrapped within the tool. Once that the Applet to be tested is wrapped within TATJA, a mechanism to uniquely identify each component is used to uniquely identify each Applet's components.

Then the user decides in which mode to operate the tool. If the tool is operated in recording mode, a recorder handler takes control of the system while if the tool is operated in playback mode, a playback handler takes control of the tool. The recorder handler and the playback handler are explained in some more detail in the next two sub sections. Note that as illustrated in Figure 5, both handlers make use of the *TatjaML document* and of the generated test classes. The *TatjaML document* is explained in detail further on in this chapter. Figure 5.5 illustrates a sequence diagram of the proposed system.

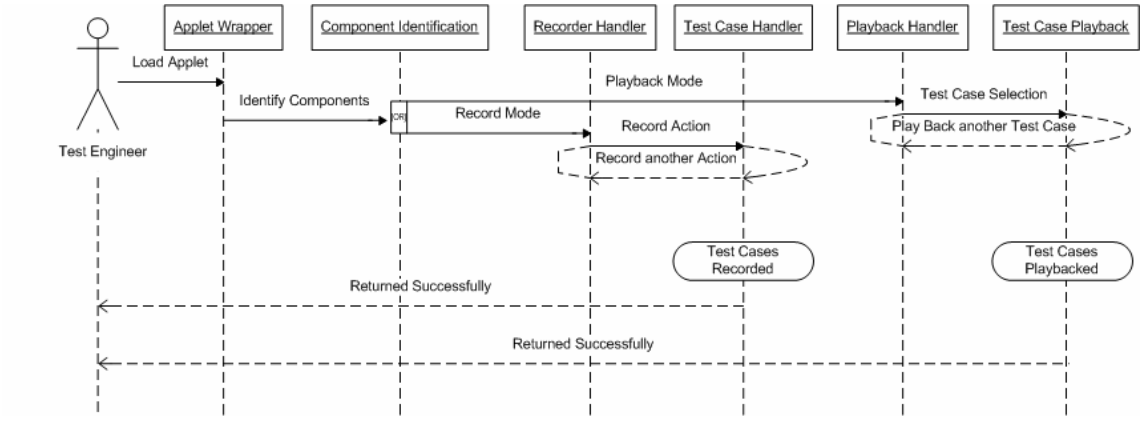

Figure 5.5: A sequence diagram of the proposed system

4.1.1 Recording Mode

Figure 6 gives an overview of the architecture of TATJA in recording mode. As shown in Figure 6, the Applet to be tested is wrapped into another Applet. The *Recorder Handler* is in charge of everything that

happens during the recording mode. In the *TatjaML File* there should be the commands together with all the required data that each Applet component can perform.

When the user switches TATJA to be used in recording mode, the *Recorder Handler* indicates the actions that the user is allowed (according to the TatjaML specification) to do. The *Recorder Handler* should handle all the actions performed by the user and all these actions are translated into compiled Java code as test cases. Note that in order to be able to record the test cases, the Recorder Handler makes use of the *Component Identification Resource*.

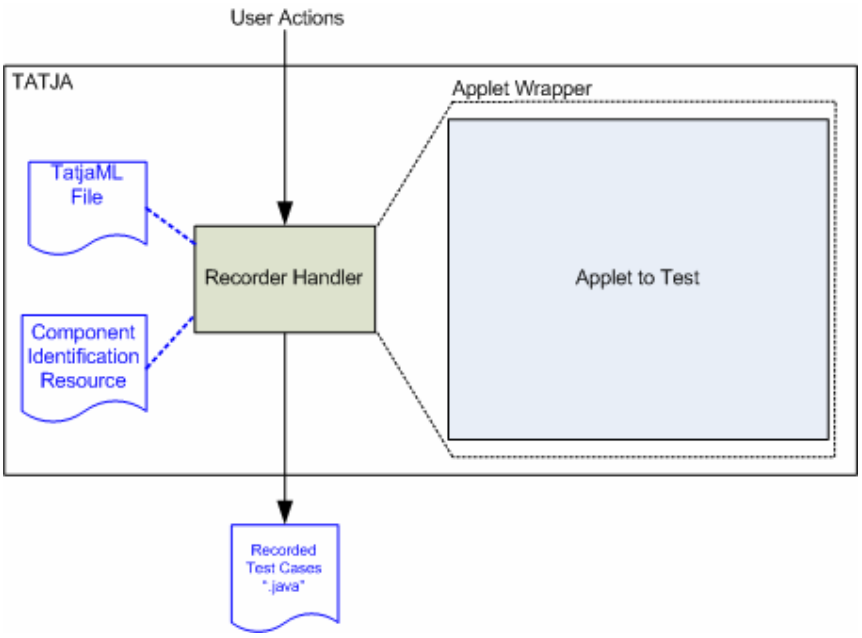

Figure 6: TATJA in recording mode

The user is allowed to record as much test cases as it is deemed necessary in different Java classes and methods. Note that in TATJA it is assumed that one method is equivalent to one single test case. A class can have many methods, i.e. many test cases.

4.1.2 Playback Mode

Figure 7 gives an overview of the architecture of TATJA when in playback mode. The Applet to be tested is wrapped within another Applet – the principle that is used in recording mode is repeated here.

When the user switches TATJA to be used in playback mode, the *Playback Handler* makes use of the *TatjaML* document in order to be able to dynamically render the actions (commands) that are coded in the test classes. Note that the *TatjaML* file to be used in playback mode should be the same that was used in the recording mode when the test cases were being recorded by the user. Note that in order to be able to playback test cases, the recorder handler makes use of the *Component Identification Resource*.

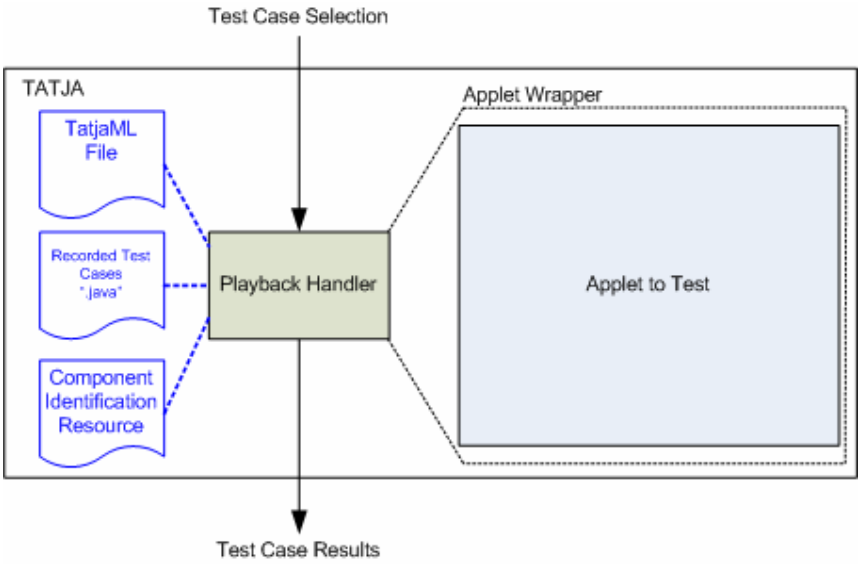

Figure 7: TATJA in Playback mode (When used manually)

The user should be allowed to playback as much test cases as it is deemed necessary. The user should also be allowed to execute the same test case over and over again.

Note that the tool should be operated in *Playback Mode* in two ways. It can be operated by executing the tool in a browser and manually selecting the test cases to be executed. This option is most likely to be used during the test cases creation phase. TATJA also has the option to programmatically execute the test cases on different web browsers.

Being able to programmatically invoke TATJA and perform tests in different web browsers, makes it possible to use and integrate the tool with other Web Application testing tools such as Selenium. This option is most likely to be used in regression testing since the test engineer can programmatically invoke a particular web browser and execute on it a selection of test cases.

Figure 8 illustrates the architecture of TATJA when it is invoked programmatically in playback mode.

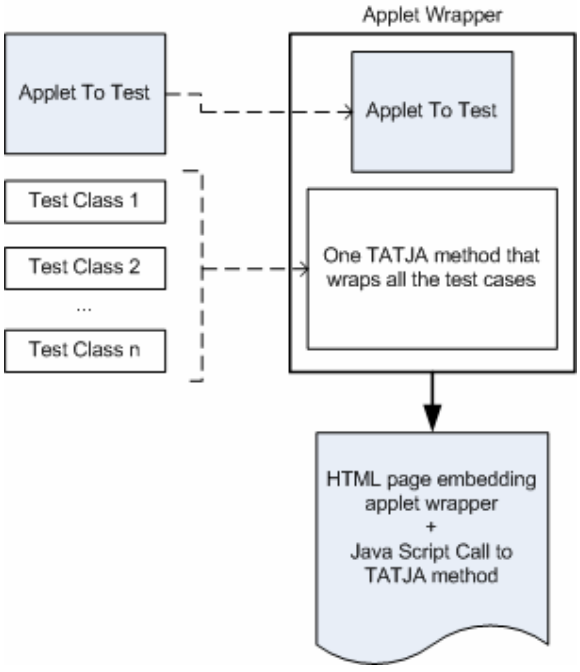

Figure 8: TATJA in playback mode (When invoked programmatically)

In the scenario depicted in Figure 8, the Applet to be tested is wrapped inside another Applet. A Java method that invokes a selection of test

classes is also generated in the Applet wrapper. This Java method is generated in order to be invoked when it is required to execute the test. An HTML page with the code that embeds the Applet is also generated. This HTML page also contains JavaScript code that invokes the previously generated Java method. When the HTML page is loaded in a web browser, the JavaScript code automatically invokes the Java method that has in it the instructions to run the desired test cases. Thus by only loading the generated HTML page in a web browser, the test cases are executed automatically.

4.2 TatjaML

TatjaML is the name of the markup language that is going to be used by TATJA. TatjaML contains a list of all the actions that can be performed on each component. An action can be either an event or a method call (command) on the component. With each action there is included all the information that is needed by TATJA to dynamically perform that operation.

```
<component name="JButton">
      <commands>
            <getName params="" desc="Description goes here">
                  <return>String</return>
                  <invoke method="getText">
                  </invoke>
            </getName>
      </commands>

      <events>
            <click params="" desc="Description goes here">
                  <listener>ActionListener</listener>
                  <invoke method="actionPerformed">
                        <pass_object object="ActionEvent">
                              <param type="java.lang.Object">^</param>
                              <param type="int">3</param>
                              <param type="java.lang.String">^getActionCommand</param>
                        </pass_object>
                  </invoke>
            </click>
      </events>
</component>
```

Figure 9: An extract taken from a *TatjaML* document

A component is defined in a TatjaML document using the *component* tag. The parameter *name* of this tag identifies the Java component (For example JPanel, JButton ...). Figure 9 shows an extract of a particular component tag taken from a TatjaML document.

A *component* tag consists of two other tags, *commands* and *events*. These two tags are used to define user actions that can be performed either by method calls or by event generation respectively. Note that each *component* definition can have zero or more *commands* and *events*.

4.2.1 Commands

A TATJA command is defined by creating a new tag within the *commands* tag. The name of this tag should be the name of the name that identifies that command. Figure 10 shows an example of a TatjaML command.

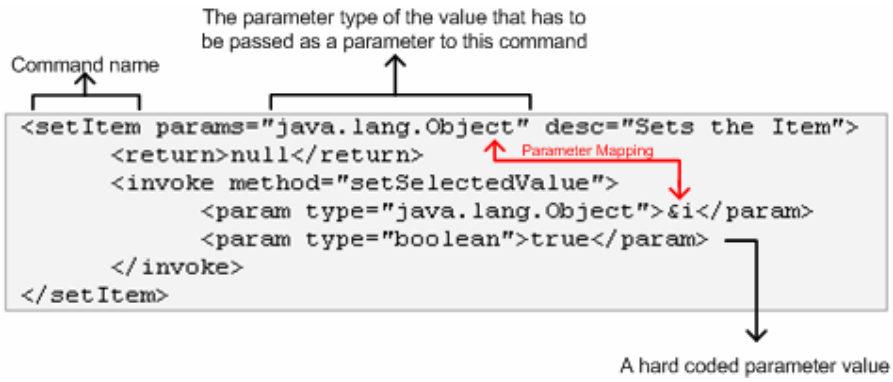

Figure 10: An example of a TatjaML command

A *command* tag has two parameters, *params* and *desc*. The parameter *params* is used to identify any parameter types (such as int, String ...) that the particular command requires when it is called by TATJA. The other parameter, *desc*, is used to let the user to write a description of that command.

The *command* tag consists of another two tags, the *return* tag and the *invoke* tag. The *return* tag is used to identify the return value of that command (such as null, int, String ...). The *invoke* tag is used to identify the method that has to be invoked on that particular component. The method name is defined in the *method* parameter.

Note that if the method requires parameters, a list of parameters are given within the *invoke* tag as *param* tags. Each *param* tag has a parameter *type* that is used to define the parameter type. The content of the *param* tag is the value of the parameter. If the value of the parameter is not constant but it has to be entered by the user as a *command* parameter, the contents of the *param* tag should be a name with an ampersand symbol (&) at the beginning (See Figure 10). An ampersand symbol at the beginning of that particular *param* tag value means that the value of that parameter has to be mapped to a parameter that was defined in the fist tag of that particular command.

4.2.2 Events

A TATJA event is defined by creating a new tag within the *events* tag. It is much the same as the *commands* tag, with the parameters *param* and *desc* for the parameters and description respectively. This type of tag is made up the tags *listener* and *invoke*. Figure 11 shows an example of a TatjaML event.

Figure 11: An example of a TatjaML event

The *listener* tag is used to identify the name of the event listener that has to be created by TATJA.

The *invoke* tag has a parameter *method* that identifies the name of the method that has to be invoked in order to generate the event. The latter has another tag, the *pass_object* tag. This tag is used to identify the event object that has to be passed as a parameter to the method that defined in the *invoke* tag. The *pass_object* tag can have zero or more *param* tags. The *param* tags are used to identify the parameters (type and value) that have to be passed to the event object in order for it to be creator.

The values of the *param* tag can be of three types. They can either be hard coded values, or values that starts with an ampersand symbol (&), or else the value can be only or starts with the carrot symbol (^). If the value starts with an ampersand symbol, then the value of that parameter has to be mapped to a parameter that was defined in the fist tag of that particular event (the same as in the *command* tag). If on the other hand the value is just the carrot symbol (See Figure 5.6), it means that that particular component has to be passed as a parameter to the event

object – self mapping. If the value starts with the carrot symbol, then the parameter to be passed is a method call on that particular event object with the method name being the string following the carrot symbol.

4.3 GUI Design

Since TATJA can be operated in two modes (Record mode and Playback mode) it is decided that the user interface of this tool should consist of two tabs – one for the record mode and the other one for the playback mode. The interface is kept as simple as possible, with only a tool bar at the top part of the Applet and the rest of the area reserved for the Applet that is being tested.

4.3.1 The Record Toolbar

Figure 12 illustrates a design of the user interface of TATJA and also the contents of the recording mode toolbar. As it is shown in the design, the *Record* toolbar consists of a number of text boxes, buttons, and icons.

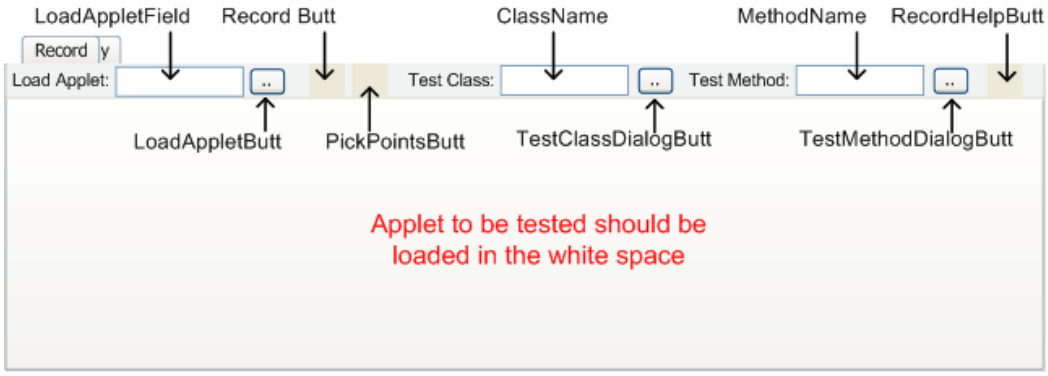

Figure 12: Design of the user interface

The fist text box is the *LoadAppletField*. This text box is used to contain the name of the Applet that is loaded in the tool in order for the

user to record tests on it. The *LoadAppletButt* button is the button that is used in order to load an Applet in TATJA. The icon named *RecordButt* is used to indicates that the uses wants to record a test case while the other icon named *PickPointsButt* is used to indicate the mouse pointer position relative to each Applet component. The *ClassName* text field s used to contain the name of the Java class where the test cases are being recorded while *TestClassDialogButt* is used to load or create a new Java test class. The *MethodName* dialog box is used to contain the name of the Java method where the tool is recording the test cases. The *TestMethodDialogButt* is used to load or create a new test method in the Java class that is listed in the *ClassName* text field.

4.3.2 The Playback Toolbar

Figure 13 illustrates a design of the user interface of TATJA and also the contents of the recording mode toolbar. This toolbar is very similar to the *Record* toolbar. The *LoadAppletField2* text field is used to store the name of the Applet that is loaded in TATJA, that is, the Applet that is being tested.

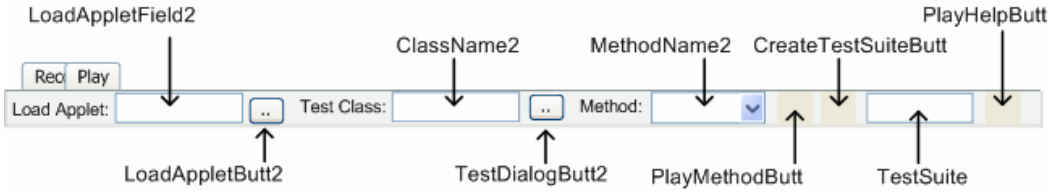

Figure 13: Design of the toolbar in playback mode

The *ClassName2* text field is used to indicate the name of the Java class where there are the recorded test cases, while the combo box *MethodName2* is used to contain the name of a particular test method that is found in the Java class listed in *ClassName2*. Note that in order to load an Applet to test or load a class that has test cases the user has to open their respective dialog boxes by clicking on the buttons *LoadAppletButt2* and *TestDialogButt2* respectively. The *PlayMethodButt* icon is used to execute the test case if the method

listed in *MethodName2*. The *CreateTestSuiteButt* icon is used to create a test suite while the *TestSuite* text box is used to contain the name of the Java class that has in it the created test suite.

4.4 Applet's Components Identification

An important operation that has to be done by TATJA is that of uniquely identifying the different components that makes up the Applet to be tested. It is important that TATJA identifies the components of a Java Applet during the play back process in the same manner that it identifies them during the recording process.

Java does not have a mechanism of uniquely identifying each component that makes up a Java Applet (or any other type of GUI). Thus it is required by TATJA to give a unique identification name to each Applet component. Note that a Java programmer can optionally give a name to a Swing component, however this name has not to be unique. Since it is not mandatory for a Java programmer to give a name for the GUI components, many programmers does not give names to the components.

In order to uniquely identify each GUI component, a hash table is used. The value of the hash table contains a reference to the actual Applet component. The key of the hash table is a generated unique identification string. If a particular component has a name (given by the programmer during the development of the Applet), than the key of that particular component in the hash table is the name given by the programmer. If there are more than one component with the same name in the same Applet, than that name is still used as an identification key however a number is appended to it in order to make it unique. If on the other hand the Applet component does not have a name, the key of that component in the hash table is the type of that component, for

example *JButton* with a number appended to it in order to make it unique, for example *JButton5*.

4.5 TATJA 'Record' Design

By having a way of uniquely identifying an Applet component, and also by having the mechanism of defining all the possible user actions on each Applet component, now it is possible to design the way that TATJA can be informed to perform an action on a particular component.

4.5.1 'Record Instruction' Design

In order to instruct TATJA to perform a particular action on a particular component, the dot notation technique is going to be adopted. Figure 14 illustrates an example of how TATJA adopted the dot notation.

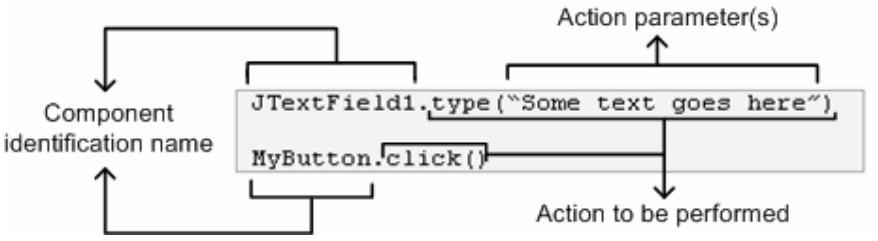

Figure 14: Example of the dot notation as adopted by TATJA

The literal before the dot is going to identify the component on which to perform the action, while the literal after the dot is going to be used to identify the action to be performed. Recall that the identification name of each component is stored in a hash table and all the possible actions that can be performed on each component are defined in the TatjaML document.

As shown in Figure 14, an action can have zero or more parameters. Since the TATJA instruction is going to be executed in the Java language, it is important that if there are any parameters within an action, the parameters are written according to the Java language specifications. That is, if a string is passed as a parameter it should be enclosed within double quotes (“”), if the parameter is a Boolean value the values can either be *true* or *false*, and if the parameter is a number it should be written according to the Java numeric types specifications.

4.5.2 ‘Record GUI’ Design

Now that the way of how to instruct a particular Applet component to perform an operation (A method call on the component or an event), is well defined, the next step is to design the GUI that the user will use in order to record a particular test. As explained in section 4.3.1 of this chapter, in order to record a test case, the user has to load the Applet to be tested in the tool, and after setting the test class and test method, the user has to click on the *RecordButt* icon to start recording.

Once the user clicks on the *RecordButt* icon, the user will be able to record actions on each of the Applet’s components. When the user moves the mouse pointer on a particular component, that particular component should be highlighted, and a popup dialog box should be displayed on this screen. This dialog box should contain all the relevant information about that component (such as type, name ...) and should also contain all the possible actins (together with their description) that are possible to do on that component. Recall that all the possible actions that can be performed on a component are defined in the TatjaML document.

For instance, suppose that the user places the mouse pointer on a *JRadioButton*, and in the TatjaML document there is the code shown in Figure 15 defined for the JRadioButton component.

```
<component name="JRadioButton">
	<commands>
		<check params="" desc="Calls method check">
			<return>null</return>
			<invoke method="setSelected">
				<param type="boolean">true</param>
			</invoke>
		</check>
		<unCheck params="" desc="Uncheck the button">
			<return>null</return>
			<invoke method="setSelected">
				<param type="boolean">false</param>
			</invoke>
		</unCheck>
	</commands>
	<events>
		<click params="" desc="Generates a click event">
			<listener>MouseListener</listener>
			<invoke method="mouseClicked">
				<pass_object object="MouseEvent">
					<param type="java.awt.Component">^</param>
					<param type="int">500</param>
					<param type="long">0</param>
					<param type="int">1024</param>
					<param type="int">^getX</param>
					<param type="int">^getY</param>
					<param type="int">1</param>
					<param type="boolean">true</param>
				</pass_object>
			</invoke>
		</click>
	</events>
</component>
```

Figure 15: Example of a TatjaML definition for a JRadioButton component

For the TATJA definition of Figure 15, the 'recording' dialog box should be displayed as sketched in Figure 16. At the top part of the dialog box there should be the details of that particular component. At the bottom part of the dialog box there should be all the possible actions as defined in the TatjaML document.

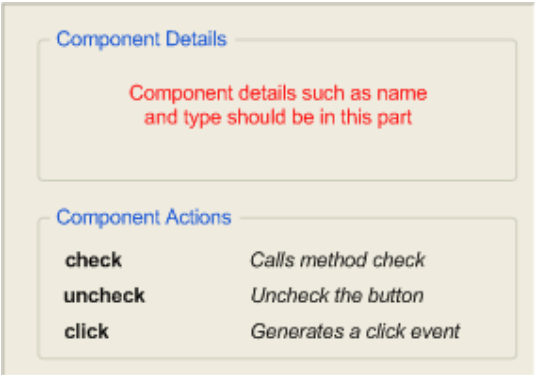

Figure 16: A sketch of the 'record' dialog box

In order to select a particular action the user has to click on the name of the desired action. If the action chosen by the user has parameters, than another dialog box should be displayed on the screen in order to let the user to enter the required parameters.

When the user records a particular action, this action is 'coded' as part of the Java method (defined in the *MethodName* text field) within the Java class (defined in the *ClassName* text field). Note that when the user removes the mouse pointer from a particular Applet component, the 'record' dialog box is disposed. If the user than places the mouse pointer on another component, a new 'record' dialog box is then generated.

4.6 TATJA 'Playback' Design

In TAJTA, a recorded test case can be executed in two ways. A test can either be executed manually from the tool itself or else it can be executed by creating a test suite that can be invoked programmatically from a Java program. In order to execute a test case manually, the user has load the Applet in the tool, select the test class and test method and click on the *PlayMethodButt* in order to start the execution of the test. Although this method is pretty easy it has the disadvantage that in order to execute a test the tool (TATJA) as to be loaded and the user

has to start the test manually. Thus it is not possible to perform regression testing. This problem is solved by creating a Test Suite.

4.7 Summary

In this chapter the architecture of the proposed tool is defined and designed. The aim for the *TatjaML document* is explained in some detail and the format of this markup language is defined. The design of the graphical user interface is also designed in this chapter. The mechanism of how the tool is going to uniquely identify each of the Applet's components is devised and the way how to record and subsequently playback test cases are devised.

Chapter 5
Implementation

This chapter discusses in some detail the main classes and algorithms that are used in TATJA.

5.1 Implementation Overview

TATJA consists of 30 Java classes and a number of configuration files. The Java classes are categorised into 5 packages according to the type of functionality they have. These packages are:

- commom
- defenitionsdriver
- dialogboxes
- helpers
- tatja

The configuration files are stored in the folder named *resources* that resides at the root of the directory of the Applet project that is being tested.

The tools used for the implementation of this project are *Eclipse 3.3.1.1 IDE* and *NetBeans 6.0 IDE*. The java runtime environment used is *jre 1.6.0 Update 5*.

5.2 Package common

This package consists of only one class that is not specific to this project, but a class that can be potentially used in other future projects that are not necessary related to software testing. This class is the *CSVTokaniser* class. It has the same functionality as the native Java *StringTokaniser*, however instead of using a space as a delimiter it uses a comma (,).

5.3 Package defenitionsdriver

This package consists of 7 classes. These are:

- CommandContainer
- DefenitionsDriver
- DetailsContainer
- EventContainer
- ParamsContainer
- RecorderToolControl
- TempCommandControl

These classes are used by TATJA in order to parse are extract information from the TatjaML document. As it was explained in the previous chapters, TatjaML is markup language that is going to be used by TATJA. As their name implies, the classes *CommandContainer, DetailsContainer, EventContainer, ParamsContainer, RecorderToolComntrol* and *TempCommandControl* are used as data place holders and does not contain a lot of functionality. However the class *DefenitionDriver* is a very critical class since it has a lot of functionality. This class and its methods are described in some detail in section 5.3.1.

5.3.1 Class DefenitionsDriver

This is a very important class since all operations related to the TatjaML document are performed through it. All the methods of this class are defined as static. This makes it possible to use this class without the need of creating an instance of it.

In the next few lines, the functionality of each method of this class is briefly explained.

public static String removeXMLComments(String s)

This method is used to remove any comments from the TatjaML document. The text of the document is passed as a parameter through s.

public static ArrayList<String> tokeniseXML(String s)

This method is used to tokenise a TatjaML file, that is, it returns the contents of the TatjaML document as a list of single elements.

public static String getXMLString()

This method is used to get the contents of the TatjaML file a one whole string.

public static ArrayList<String>
getEventsOfComponent(ArrayList<String> tokens)

This method is used to get a list of all the defined events of a particular Applet component. The list of tokens of that particular component is passed as parameter.

public static ArrayList<String>
getCommandsOfComponent(ArrayList<String> tokens)

This method is used to get a list of all the defined commands (method calls) of a particular Applet component. The list of tokens of that particular component is passed as parameter.

public static ArrayList<TempCommandContainer>
getControlDetailsOfComponent(String componentName)

This method is used to get a list of the required control information of a particular component. The name of the component is passed as parameter.

public static String getCommandDescription
(ArrayList<String>tokens,String command)

This method is used to get the description of a particular command that is defined in the TatjaML document. The list of tokens of the particular Applet component and the name of the command are passed as parameters.

public static boolean isCommandEvent
(ArrayList<String> componentTokens,String componentName,String command)

This method is used to check if a particular defined action of a particular component is an event or not. The list of tokens of the particular TatjaML document, the name of the component and the name of the command are passed as parameters.

public static String getTagParameter(String tag,String parameter)

This method is used to get the parameter of a particular TatjaML tag. The tag and the name of the parameter are passed as parameters.

public static ParamsContainer getCommandParams
(String componentName,String command)

This method is used to get the parameter details of a particular component command through an instance of a *paramsContainer* class. The name of the component and name of the command are passed as parameter.

public static ArrayList<String> getTokensOfTag
(ArrayList<String> tokens,String tagName)

This method is used to tokenise a particular TatjaML tag detention. The list of tokens of the particular Applet component and the tag name are passed as parameters.

public static String getTagArgument(String tag)

This method is used to get the argument of a particular TatjaML tag. The tag content is passed as parameter.

public static String getTagName(String tag)

This method is used to get the name of the TatjaML tag that is passed as a parameter.

public static ArrayList<String> getTokensOfComponent
(ArrayList<String> list,String componentName)

This method is used to get the list of token elements of a particular Applet component that is defined in the TatjaML document. This list of tokens of the whole TajtaML document and the component name are passed as parameter.

public static ArrayList<String>
getCommandNames(ArrayList<String> t)

This method is used to get the list of command names of a particular Applet component as defined in the TatjaML document. The list of tokens of that particular component is passed as parameter.

public static ComponentCommands getComponentCommands
(ArrayList<String> list,String componentName)

This method is used to get the all commands (as defined in the TatjaML document) together with their details of a particular component through an instance of a *CompoentCommands* class.

The list of tokens of the whole TatjaML document and the component name are passed as parameters.

public static CommandContainer getCommandDetails
(ArrayList<String> tokens,String commandName)

This method is used to get the details of a particular component command through an instance of class *CommandContainer*. The list of tokens of the particular component and the command name are passed as parameters.

public static EventContainer getEventDetails
(ArrayList<String> tokens,String eventName)

This method is used to get the details of a defined event in the TajtaML document of a particular component through an instance of class *EventContainer*. The list of tokens of the particular component and the name of the event are passed as parameters.

public static ArrayList<String> getListOfAllComponents
(ArrayList<String> tokens)

This method is used to get a list of all the defined components in the TatjaML document. The list of tokens of the whole TatjaML document is passed as parameter.

5.4 Package dialogboxes

This package consists of 13 Java classes. These are:

- AssertDialogBox
- CommandConstructionDialogBox
- DelayDialogBox
- LoadAppletDialogBox
- LoadTestSuiteDialogBox

- NewTestSuiteDialogBox
- PlayClassDialogBox
- PointPickerDialogBox
- PreDefinedCommandsPanel
- RecorderToolPanel
- TestCkassChooserDialogBox
- TestMethodChooserDialogBox
- TextWindow

As the name of the package and of the classes themselves implies, the role of these classes is to provide the required dialog boxes to TATJA. These dialog boxes are implemented using the Java Swing API.

5.5 Package helpers

Package *helpers* consist of 4 Java classes. As the name of the package implies, the classes that are stored in this package are helper classes to for TATJA. These classes are:

- ComponentCommands
- Miscellaneous
- TATJAConsole
- TestHelpers

The class *ComponentCommands* is a simple information place holder class. It is used to store the all the commands together with their required details of a particular component. This class makes use of a hash table in order to map the command name with a reference to a *DetailsContainer* object.

The class *Miscellaneous* has a number of static methods that perform various simple but vital operations. Such operation includes appending

instruction to generated test cases, creation of new Java methods in a test class, and more.

The class *TATJAConsole* is a class that has only 2 static methods. These are:

- **public static void runFileInWebBrowser (String browserConstant,String filePath)**
- **public static void runHTTPPageInWebBrowser (String browserConstant,String url)**

These 2 methods are used to run an HTML page in a specific web browser by either specifying the URL or else specifying the absolute file path of where there is the HTML document. The string parameters *browserConstant* are there to define the name/type of browser to invoke such as 'Internet Explorer 6.0' or 'Opera' or 'Safari'. Note that in order to be able to use these two methods, some configuration details have to be set up in the configuration file *WebBrowserConfigs.properties*. More on the configuration files in section 5.7 of this chapter.

The *TestHelpers* class is a fundamental class in TATJA. This class and some of its public methods are described in some details in section 5.5.1.

5.5.1 Class TestHelpers

Class *TestHelpers* is a fundamental class because it is used to convert the commands and actions defined in the TatjaML document into real Java actions dynamically. As it was explained earlier in Chapter 3, this is done by using the Java Reflection API.

This class consists of a number of private and public methods, however the most two fundamental methods are *invokeCommand* and *invokeEvent*. These methods are explained in some detail below.

5.5.1.1 Method invokeCommand

```
public Object invokeCommand(Object component, String invoke,
                            String returnType, String[] params,
                            String[] args, String commandParameters,
                            String commandArguments)
```

This method is used to invoke a particular command on a particular Java Applet component. The Applet component on which to invoke the command is passed as a parameter through the argument *component*. The name of the java method to invoke in order to execute the command is passed as a string parameter through *invoke* while the return type of the method is passed as a parameter through argument *returnType*.

The arguments of the Java methods are passed a string array parameter through argument *args*. If there are any parameters that are passed by the user, the arguments are defined as a comma separated string and passed as a parameter through the argument *commandArguments*. The respective values of the latter are passed as a comma separated string parameter through argument *commandParameters*.

If the invoked command requires returning a value, the value is returned through the generic class type *Object*. The code snippet below shows the computation that is performed by this method in order to invoke a command (method) on a particular component. Inline comments are included within the snippet for code clarifications.

```
public Object invokeCommand(Object component,String invoke,String
returnType,String[] params,
String[] args,String commandParameters,String commandArguments){
      Object ans = null;
      try{
            //First create a class instance of the object component to
            //invoke command on
            String className = component.getClass().toString();
            className = className.substring(className.lastIndexOf(" ") +
                                          1,className.length());
            Class c = Class.forName(className);
            //Create the method to invoke
            Method m = c.getMethod(invoke,getParametersClasses(params));
            Object[] arguments =
                        getConstructorArguments(component,args,params);
            //Merge arguments[] and args[]
            Object[] test = new Object[args.length];
            for(int i = 0;i < test.length;i++){
                  if(arguments[i] == null){
                        test[i] = args[i];
                  }else{
                        test[i] = arguments[i];
                  }
            }
            arguments = fixArguments(test,commandArguments,params);
            ans = m.invoke(component,arguments);
      }catch(Exception e) {
            e.printStackTrace();
      }
      return ans;
}
```

Code Snippet of method *invokeCommand* from class *TestHelpers*

5.5.1.2 Method invokeEvent

public void invokeEvent(Object component,String listener,
String invoke,String passObject, String[] params,
String[] args,String eventParameters, String eventArguments)

This method is used to create and invoke a particular event on a particular Java Applet component. The Applet component is passed as parameter through argument *component*. In order to generate a Java event, an event listener has first to be initiated. The name of this event is passed as parameter through *listener* while the name of the method to invoke on the event listener is passed through argument *invoke*.

The name of the event object that is passed to the component is passed as parameter through the argument *passObject*. A string array with the arguments to pass to the event object is passed as parameter through *args* while their respective values are passed through *params*. If there are any parameters that are passed by the user, the arguments are defined as a comma separated string and passed as a parameter through the argument *eventArguments*. The respective values of the latter are passed as a comma separated string parameter through argument *eventParameters*.

The code snippet below shows the computation that is performed by this method in order to create and invoke an event on a particular component. Inline comments are included within the snippet for code clarifications.

```
public void invokeEvent(Object component,String listener,String
invoke,String passObject, String[] params,String[] args,String
eventParameters,String eventArguments){
      try{
            //First create object to get
            Class event = Class.forName("java.awt.event." + passObject);
            //Get constructor parameters
            Class[] constParam = getParametersClasses(params);
            //Get constructor arguments
            Object[] arguments =
                        getConstructorArguments(component,args,params);
            arguments = fixArguments(arguments,eventArguments,params);
            //Create event constructor
            Constructor constructor = event.getConstructor(constParam);
            //Create an instance of the event and pass to it the
            //constructor arguments
            Object object = constructor.newInstance(arguments);
            // Create a listener type
            Class c = Class.forName("java.awt.event." + listener);
            //Get an array with all the listener types of the component
              //...first create a class type
              String className = component.getClass().toString();
              className = className.substring(className.lastIndexOf(" ") +
                                                1,className.length());
              Class temp = Class.forName(className);
              //...then create the method to get the listeners
              Method method = temp.getMethod("get" + listener + "s",null);
            EventListener[] listeners =
                        (EventListener[])method.invoke(component,null);
            // Create the event method to invoke
            method = c.getMethod(invoke,event);
            // Invoke the method on each event listener
            for(int i = 0;i < listeners.length;i++){
                  method.invoke(listeners[i],object);
            }
      }catch(Exception e){
            e.printStackTrace();
      }
}
```

Code Snippet of method *invokEvent* from class *TestHelpers*

5.6 Package tatja

Package *tatja* consists of 4 classes. These are:

- AppletTester
- Constants
- RecorderHandler
- TestPlayHandler

As its name implies, class *Constants* is a simple class with some constants detentions. However the other three classes are fundamental classes and can be defined as the 'heart' or the engine that drives the whole test automation tool. These classes are defined in the next three sections.

5.6.1 Class AppletTester

Class *AppletTester* is the main class of this test automation tool. As explained in the previous chapters, it is a Java Applet that wraps in it another Applet – the Applet to be tested.

Apart from providing the wrapping mechanism and the GUI of TATJA, this class performs a lot of functionality. All the event handling of the tool is performed by this class. Figure 17 shows a screen shot of TATJA when executed using *Sun's Applet Viewer*. At the top part of the application there is the tabbed toolbar. One of the tabs is for recording mode while the other one is for playback mode. The rest of the area is left so that the Applet to be tested can be loaded there.

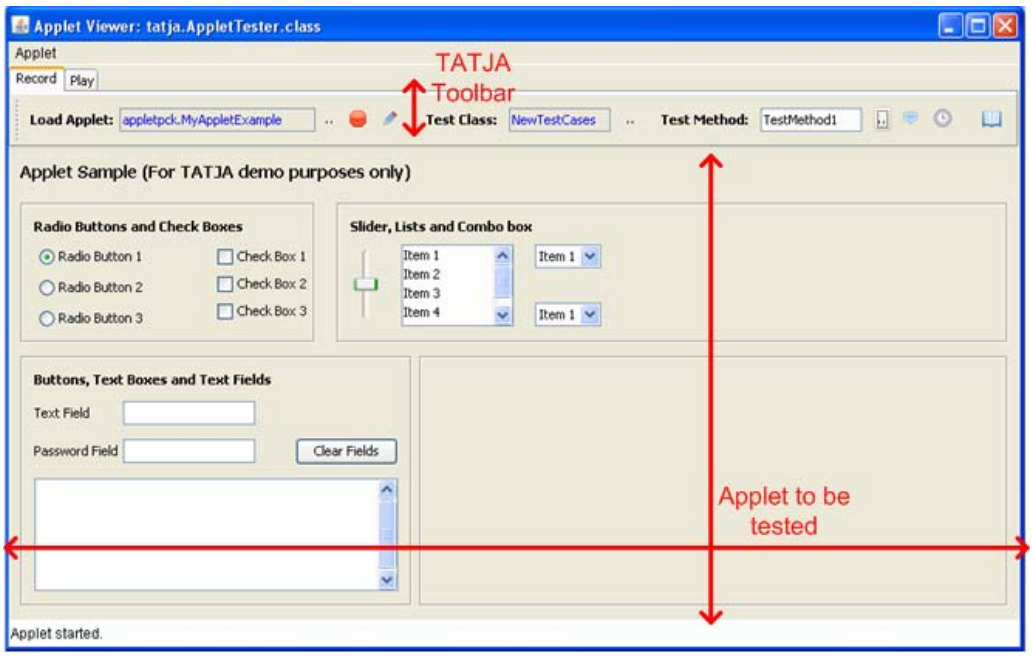

Figure 17: A screen shot of TATJA in Sun's Applet Viewer

5.6.2 Class RecorderHandler

When the user clicks on the record icon, the *RecorderHandler* comes into action. This class is used by TATJA in all the recording process. When this class is initiated, it first creates a hash table with all the components that are part of the Applet to be tested. The components are given a unique identification name. If the components do not have an identification name, they are given an identification name by the order of how the components were added to the Applet to be tested during its development process.

Class *RecorderHandler* takes also the control of the tool's GUI interface during the recording process. During this process, this class manipulate some of the component's properties in order to make them highlight when the user is recording a particular action on them. The *RecorderHandler* class has also methods that make use of the *DefenitionsDriver* class that is defined in the *defenitionsdriver* package. These methods are able to get all the names of the actions that can be performed on each particular component together with the necessary details of each action. These details are displayed in a tailor made dialog box from which the user can decide which action to record when the user places the mouse pointer on a particular component as shown in Figure 18.

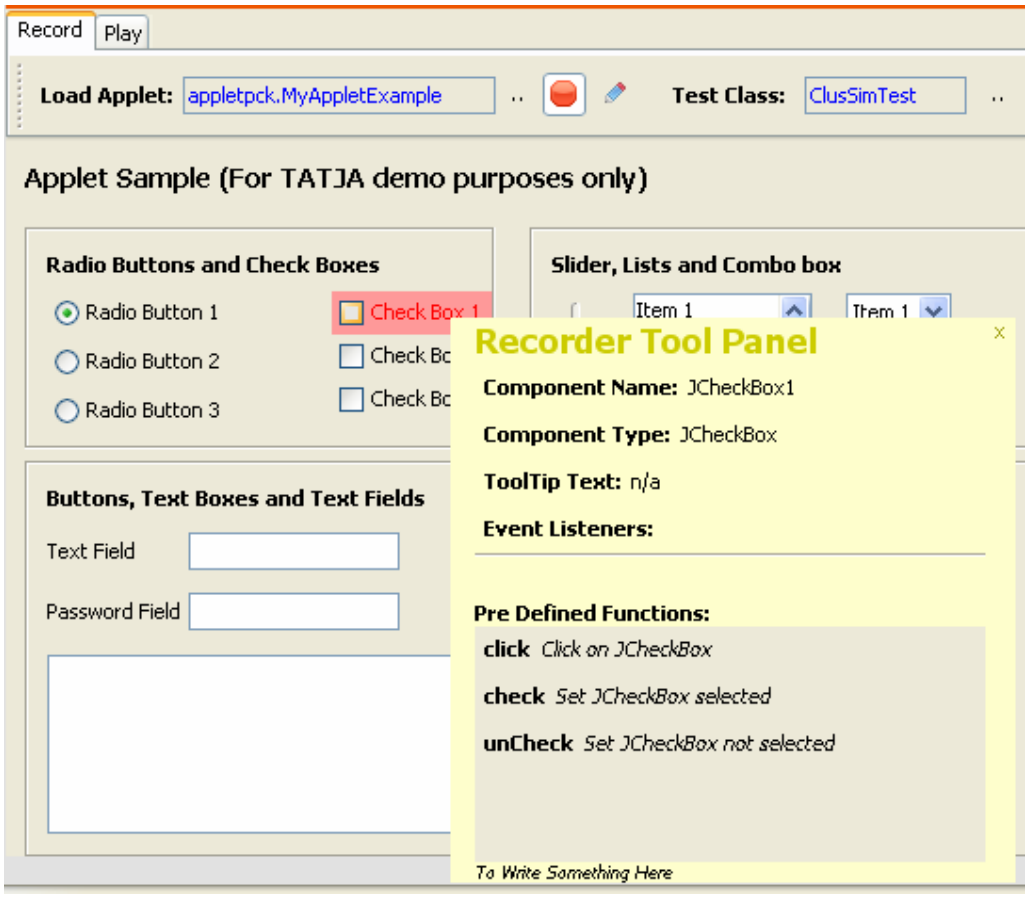

Figure 18: A screen shot of TATJA with the mouse pointer on a component and with the Record Dialog Box Visible

The user can click on any of the available actions of each component and *RecorderHandler* will generate all the necessary Java code. This generated code can than be executed and play back in order to run the recorded test cases by using the other class of the *tatja* package – the *TestPlayHandler* class.

5.6.3 Class TestPlayHandler

Class *TestPlayHandler* is used in order to play back the recorded test cases. When this class is initiated, it first identifies and stores in a hash table uniquely each component that makes up the Applet to be tested. By having a hash table that maps each unique name to its component, by using some of its methods, *TestPlayHandler* can execute

and playback the recorded test cases as if executing a conventional Java method.

5.7 TATJA Configuration files

TATJA requires 3 configuration files in order to be able to operate correctly. These configuration files should be places in the folder named *resources* at the root of the directory of the Applet project that is being tested. These configuration files are:

- DefaultComponents.csv
- Definitions.xml
- WebBrowserConfigs.properties

The *DefaultComponents.csv* file is a file that has a list (comma separated) of all the Java Swing native components such as *JButton*, *JTextField*, and more. The reason behind this configuration file is so that TATJA will know if a particular component is a native Swing component or else it is a tailor made component extending from a native one.

The *Definitions.xml* file is a very important file. This file has in the all the TatjaML definitions. Without these definitions TATJA will not be able to record any test cases.

The last configuration file is the *WebBrowserConfigs.properties*. This configuration file is used by the two methods of class *TATJAConsole*. Since these two methods invokes different web browsers, the shell commands that invokes and loads the browsers (Operating System specific) are stored in this configuration file.

5.8 Summary

In this chapter the implementation details of the system are discussed. A description is given for each package. A description is also given to some fundamental classes and also to some important methods. An explanation is given for each configuration file that is used by TATJA.

Chapter 6
Testing and Evaluation

6.1 Testing Strategy

White box and *black box* testing are applied on TATJA. Each component of TATJA is developed and tested on its own. Where applicable unit tests are carried out.

Integration testing follows after component testing to test if the different modules still function correctly after being interfaced with each other. Finally system and acceptance testing are carried out to check if the system has met the specifications it was designed for and to see if the system is performing well over-all.

After the testing process, three test cases on real Applets are performed to evaluate the system. These three Applets are:

1. A Data Clustering Simulation Applet
2. A Car Loan Calculator Applet
3. A Frictionless Slope Simulation Applet

In the next three sections of this chapter each test case is explained in some detail.

6.2 Test Case 1 – A Data Clustering Simulation Applet

The *Data Clustering Simulation Applet* is a Java Applet that is used to simulate a number of data clustering methods. The user creates a number of nodes by clicking on the grey panel. After selecting the type of clustering algorithm to use, the user should click on the start button to start the simulation process. Each time that the user clicks on the next button the Applet groups a new cluster until all the created nodes are clustered into one group. Figure 19 is a screen shot of this Applet in *Sun's Applet Viewer Application*.

Figure 19: A Screen Shot of the Applet to be tested in Sun's Applet Viewer

6.2.1 Test Case Design

6.2.1.1 Record Outline

1. Load TATJA in FireFox
2. Load the Applet to test in TATJA (Recording Mode)

3. Create a test class – *ClusSimTest*
4. Record the following test case in a method named *TestCase1*:
    i. Generate 10 random nodes in the grey panel.
    ii. Select a *Single Agglomerative Clustering* as clustering algorithm from the clustering method combo box.
    iii. Generate a click event on the *Start* button, and then subsequently on the *Next* button until all the nodes are clustered into one whole group.

6.2.1.2 Playback Outline:

1. Load TATJA in FireFox
2. Load the Applet to test in TATJA (Playback Mode)
3. Load the class and method with the recorded test case.
4. Playback the test case.
5. Repeat the playback process on Safari.

6.2.2 Test Case Execution

As it is outlined in section 6.2.1 TATJA is executed in Firefox and the Applet to be tested is loaded. A new test class named *ClusSimTest* and a new test case (method) *TestCase1* are created as shown in Figure 20.

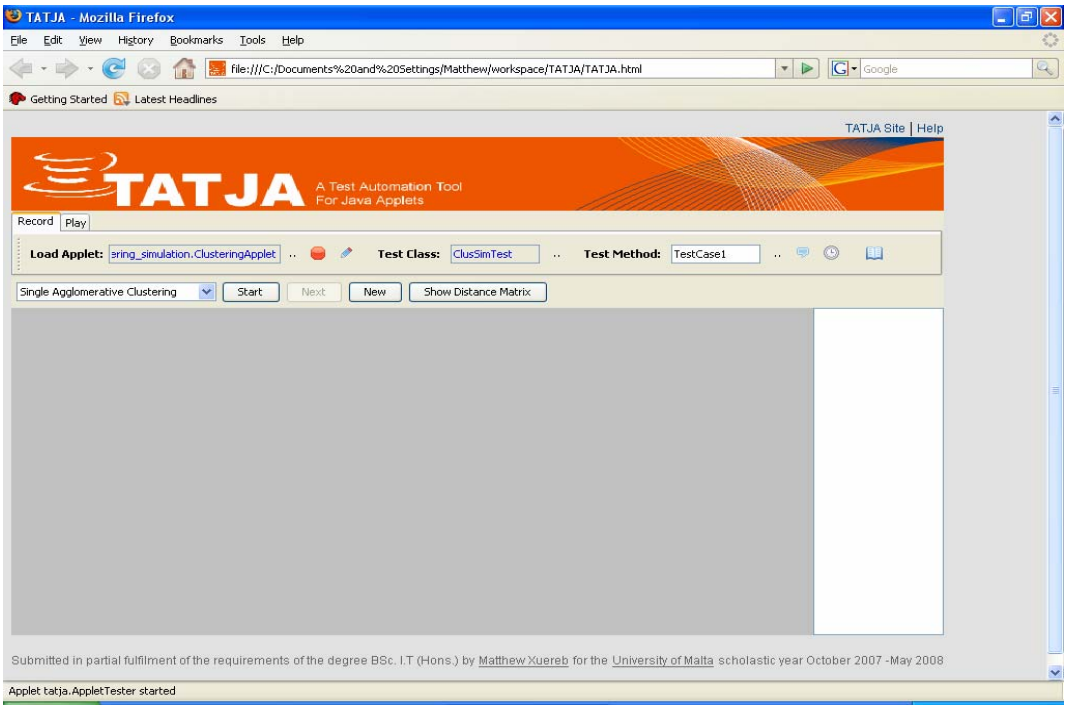
TATJA - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Documents%20and%20Settings/Matthew/workspace/TATJA/TATJA.html
Getting Started Latest Headlines
TATJA Site | Help
TATJA
A Test Automation Tool
For Java Applets
Record Play
Load Applet: sring_simulation.ClusteringApplet
Test Class: ClusSimTest
Test Method: TestCase1
Single Agglomerative Clustering
Start
Next
New
Show Distance Matrix
Submitted in partial fulfilment of the requirements of the degree BSc. I.T (Hons.) by Matthew Xuereb for the University of Malta scholastic year October 2007 -May 2008
Applet tatja.AppletTester started

Figure 20: A screen shot with the *Clustering Simulation Applet* loaded in TATJA on Firefox

The test case is recorded as outlined in 6.2.1.1 and the required code is generated as shown in the code snippet below:

```
public class ClusSimTest extends tatja.TestPlayHandler{

   public ClusSimTest(JApplet test,Map<String,Component> map){
      super(test,map);
   }

   public void TestCase1(){ // TATJA Generated TestCase1

         Random randomGenerator = new Random();

         for(int i = 0;i < 10;i++){
               String s = "ClustersPanel1.singleClick(" +
                                 (randomGenerator.nextInt(770) + 1) +
                                 "," +
                                 (randomGenerator.nextInt(345) + 1) +
                                 ")";
               runCommand(s);
         }
        runCommand("JButton1.click()");
        while((Boolean)runCommand("JButton2.isClickable()")){
          runCommand("JButton2.click()");
        }
   }// TATJA end of TestCase1
} // End of TATJA generated class
```

The recorded test case was play backed both on Firefox and on Safari. Figure 21 illustrates the executed test case on Safari.

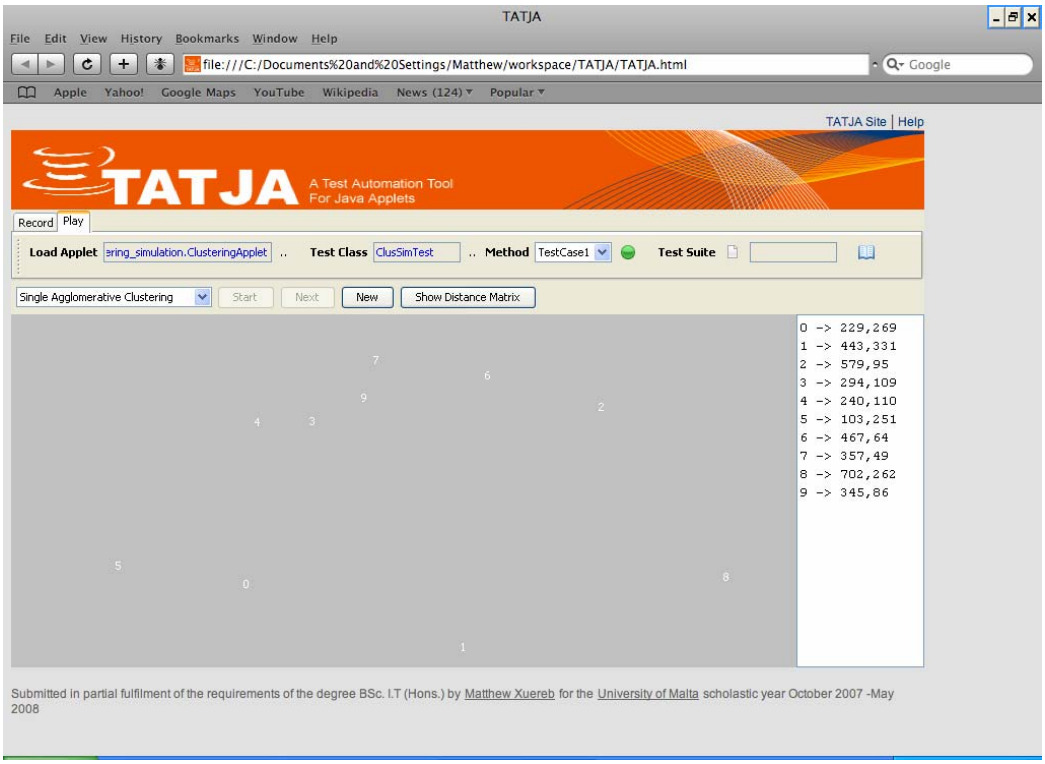

Figure 21: A screen shot of the executed test case on Safari

6.3 Test Case 2 – A Car Loan Calculator Applet

The *Car Loan Calculator* is a simple Applet that is used to calculate all the relevant details that are required when taken a loan in order to purchase a car. This Applet and its code are available online [24]. Figure 22 illustrates a screen shot of this Applet as viewed on Sun's Applet Viewer.

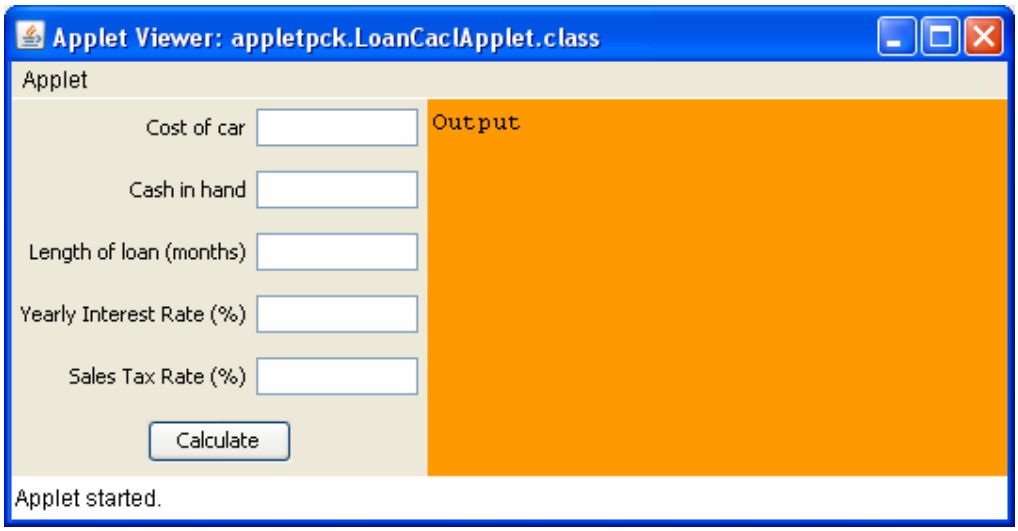

Figure 22: A screen shot of the *Car Loan Applet Calculator* as executed in Sun's Applet Viewer

6.3.1 Test Case Design

6.3.1.1 Record Outline

1. Load TATJA in Opera
2. Load the Applet to test in TATJA (Recording Mode)
3. Create a test class – *LoadCalcTest*
4. Record the following test case in a method named *TestCase1*:
    i. Set the following data and then click on calculate:

Cost of Car	$ 4535
Cash in hand	$ 3465
Length of Loan	24 Months
Yearly Interest Rate	4.6%
Sales Tax Rate	1.0%

6.3.1.2 Playback Outline:

1. Load TATJA in Opera
2. Load the Applet to test in TATJA (Playback Mode)
3. Load the class and method with the recorded test case.
4. Playback the test case.
5. Repeat the playback process on Internet Explorer.

6.3.2 Test Case Execution

As outlined in section 6.3.1, the Applet is loaded in TATJA on Opera as a web browser. A test class named *LoanCalcTest* and test case named *TestCase1* are created as shown in Figure 23 below.

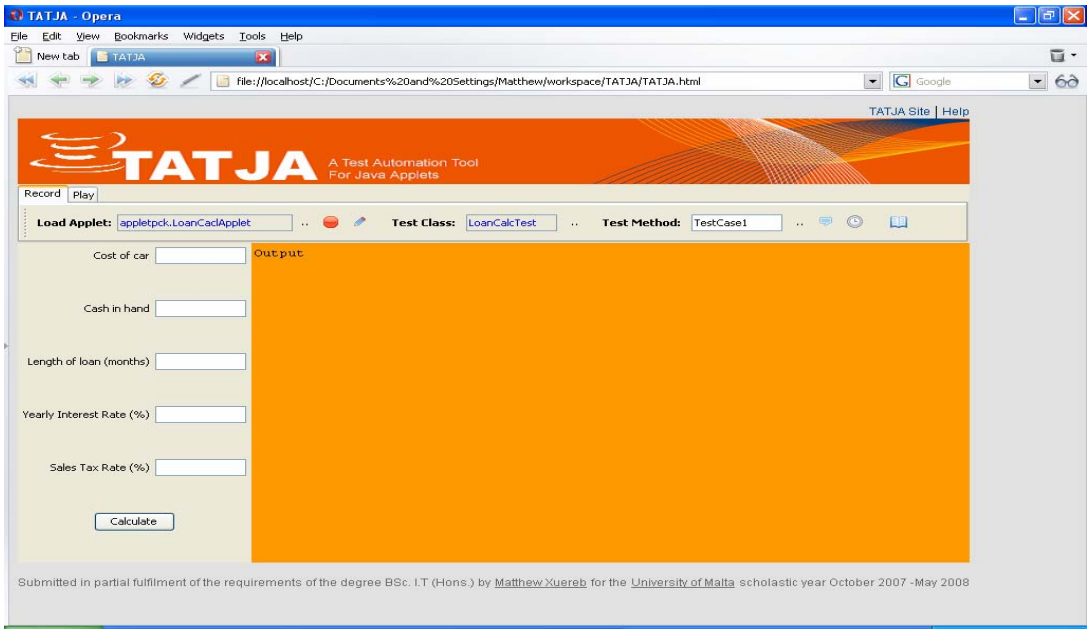

Figure 23: A screen shot with the *Car Loan Calculator Applet* loaded in TATJA on Opera

The test case is recorded as outlined and the required code is generated as shown in the code snippet below:

```
public class LoanCalcTest extends tatja.TestPlayHandler{

   public LoanCalcTest(JApplet test,Map<String,Component> map){
      super(test,map);
   }

   public void TestCase1(){ // TATJA Generated TestCase1

      runCommand("JTextField1.type(4535)");
      runCommand("JTextField2.type(3465)");
      runCommand("JTextField3.type(24)");
      runCommand("JTextField4.type(4.6)");
      runCommand("JTextField5.type(1.0)");
      runCommand("JButton1.click()");
   }// TATJA end of TestCase1
} // End of TATJA generated class
```

The recorded test case was play backed both on Opera and on Internet Explorer. Figure 24 illustrates the executed test case on Opera.

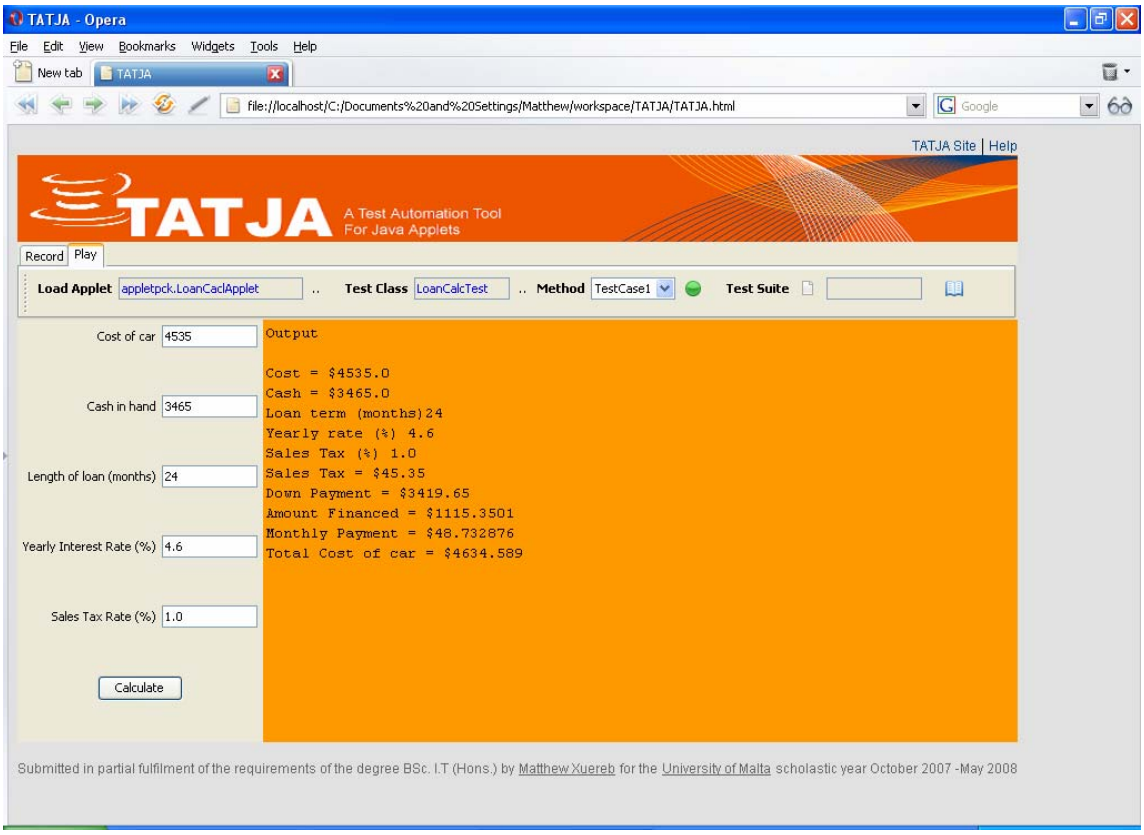

Figure 24: A screen shot of the executed test case on Opera

6.4 Test Case 3 – A Frictionless Slope Simulation Applet

The *Frictionless Slope Simulation Applet* is one of a serious of simulations Applets available on Mr Damien Martin's (University of California at Davis) website [25]. The Applet and its code are available on [26]. It is used to by students to show the physics behind an object moving down a slope. Figure 25 illustrates a screen shot of this Applet as shown on Sun's Applet Viewer.

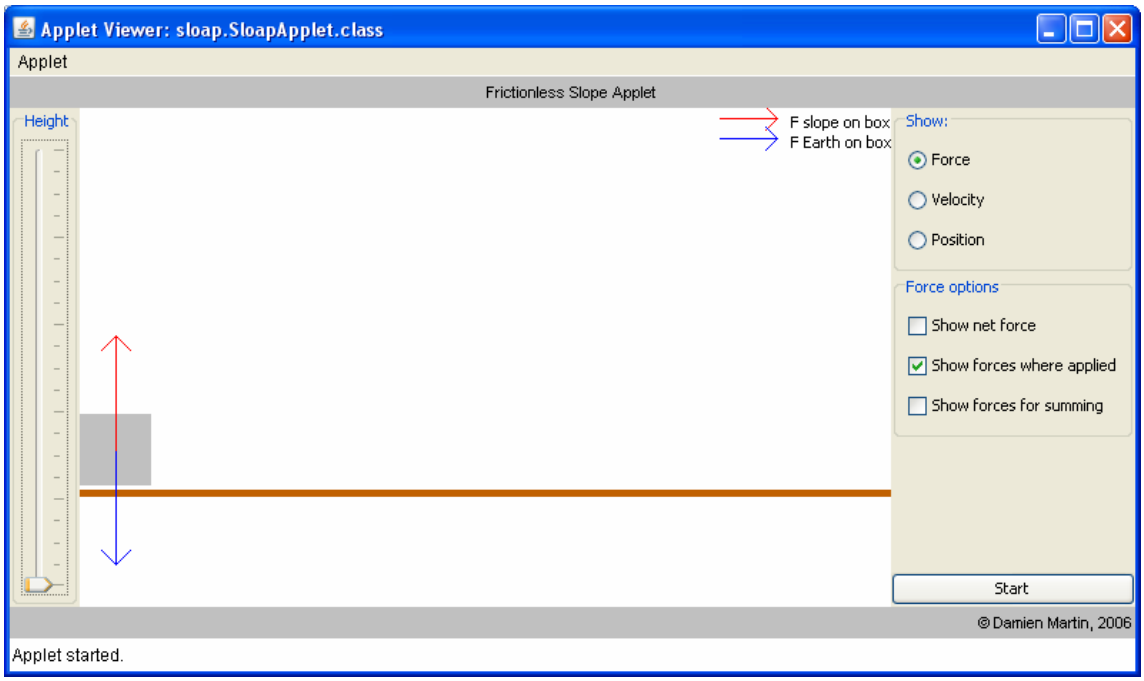

Figure 25: A screen shot of the *Frictionless Slope Applet* as executed in Sun's Applet Viewer

6.4.1 Test Case Design

6.3.1.1 Record Outline

1. Load TATJA in Netscape Navigator
2. Load the Applet to be tested in TATJA (Recording Mode)
3. Create a test class – *SlopeTest*
4. Record the following test case in a method named *TestCase1*:
    i. Set to slope to be at 30%.
    ii. Set the 'show net force' and 'show forces for summing' check boxes selected.
    iii. Start the simulation.
    iv. After 3 seconds set the velocity radio button.
    v. After 3 seconds set the position radio button.
    vi. After 5 seconds click on the reset button.

6.3.1.2 Playback Outline

1. Load TATJA in Netscape Navigator.
2. Load the Applet to test in TATJA (Playback Mode).
3. Load the class and method with the recorded test case.
4. Playback the test case.
5. Create a simple Test Suite to be able to automatically execute the generated test case.
6. Execute the Test Suite on Internet Explorer and Natscape Navigator.

6.4.2 Test Case Execution

The Applet is loaded on Netscape Navigator as a web browser. A test class named *SlopeTest* and test case named *TestCase1* are created as shown in Figure 26 below.

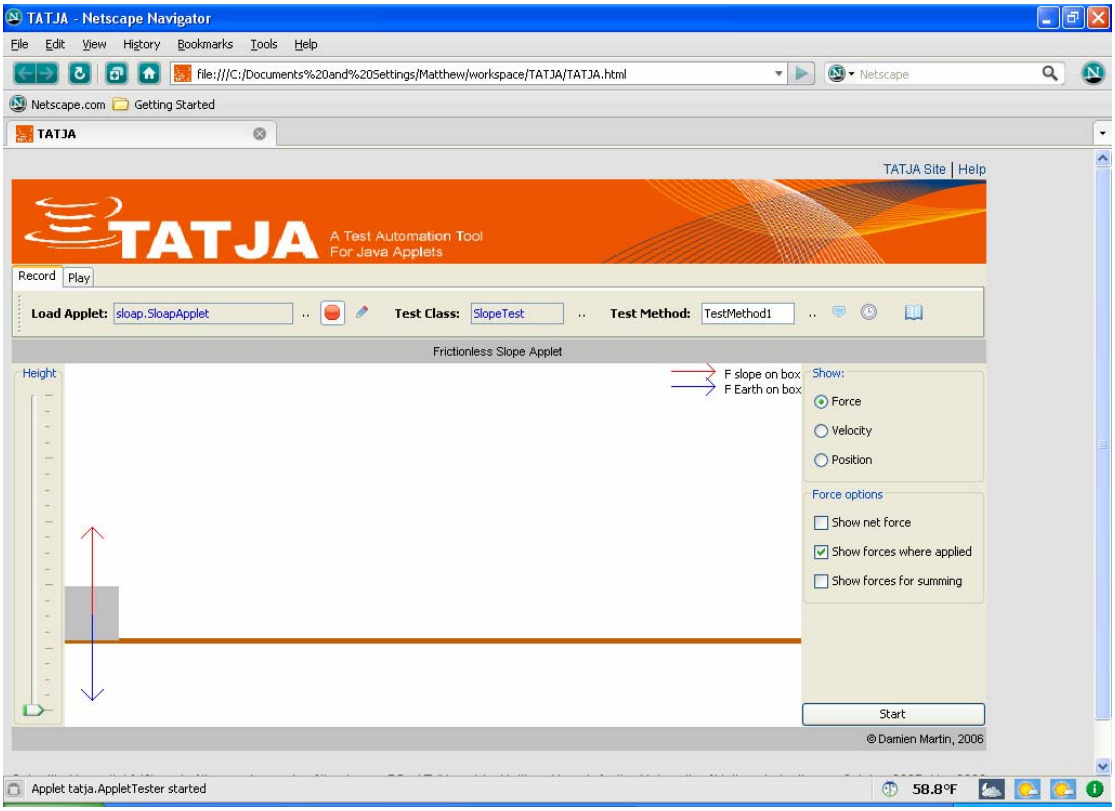

Figure 26: A screen shot with the *Frictionless Slope Applet* loaded in TATJA on Netscape Navigator

The test case is recorded as outlined and the required code is generated as shown in the code snippet below:

```
public class SlopeTest extends tatja.TestPlayHandler{

   public SlopeTest(JApplet test,Map<String,Component> map){
      super(test,map);
   }

   public void TestMethod1(){ // TATJA Generated TestMethod1

      runCommand("JSlider1.setPosition(30)");
      runCommand("JCheckBox1.click()");
      runCommand("JCheckBox3.click()");
      runCommand("JButton1.click()");
      try{
          Thread.sleep(1000 * 3);
      }catch(InterruptedException e){
      }

      runCommand("JRadioButton2.click()");
      try{
          Thread.sleep(1000 * 3);
      }catch(InterruptedException e){
      }

      runCommand("JRadioButton3.click()");
      try{
          Thread.sleep(1000 * 5);
      }catch(InterruptedException e){
      }

      runCommand("JButton1.click()");
   }// TATJA end of TestMethod1
} // End of TATJA generated class
```

The recorded test case is play backed on Netscape Navigator. Figure 27 shows a screen shot of the executed test case.

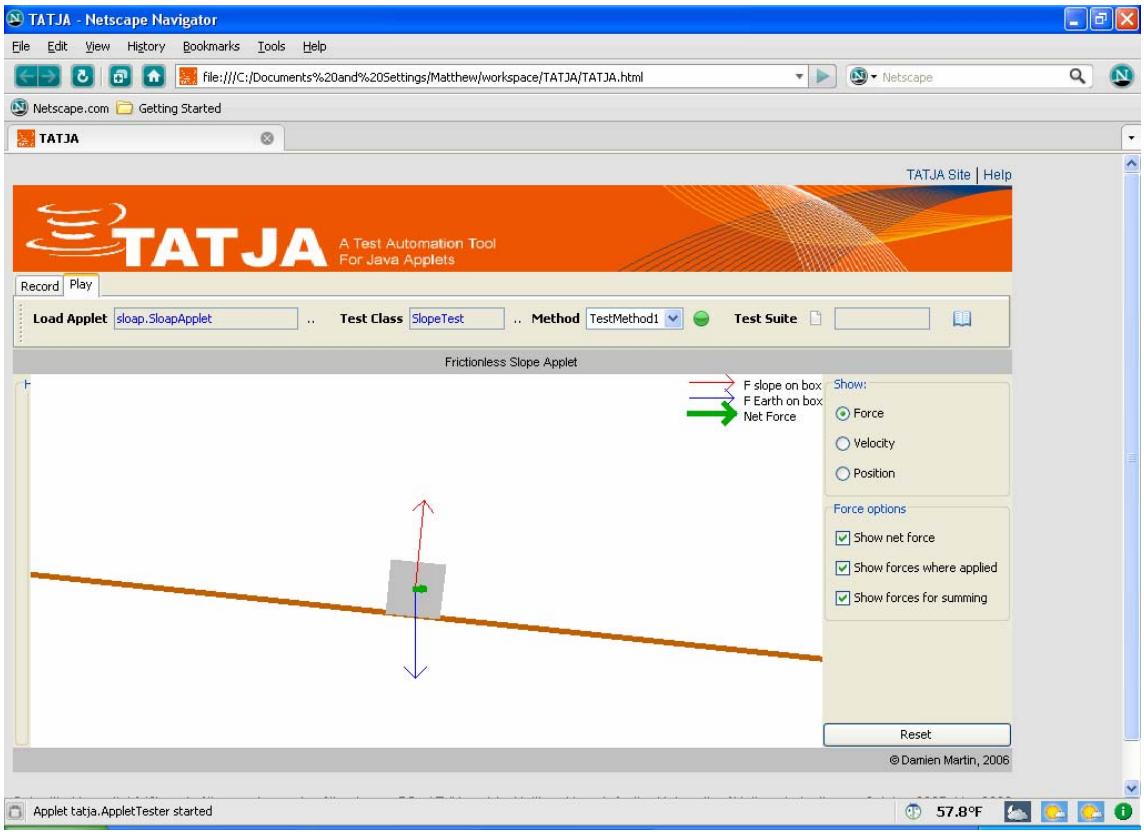

Figure 27: A screen shot of the executed test case on Netscape Navigator

After the above is executed, a test suite of the above test case is created and executed on Internet Explorer and Netscape Navigator.

6.5 Summary

In this chapter a brief description of the testing carried out on the tool is given. The tool is tested with three different working Applets. The results of the three test cases performed on these Applets are reported in this chapter. The test cases were executed on four different web browsers: Microsoft Internet Explorer, Mozilla Firefox, Safari, Netscape Navigator and Opera.

Chapter 7
Conclusions and Future Work

This dissertation is concluded by discussing the results obtained in the previous chapter – *Testing and Evaluation*. In view of these results, some improvements on the technique used and some suggestions are presented.

7.1 Summery of Results

The tool has been tested with existing Java Applets of different types. In Chapter 6, TATJA was tested on three working Applets that between them they cover the type of components that are used in Java Applets. The testing tool was also tested on various web browsers to make sure that there are no browser issues that can make TATJA not performing as expected.

The desired functionality of the tool was obtained. The testing tool also gave the expected results for all the test cases carried out. It can thus be concluded that all objectives set for this project were successfully achieved.

7.2 Future Work and Research

Although the tool was successfully implemented and all the objectives set for this project were successfully achieved, obviously there is room for improvement in order to make the tool more usable. With the current state of the tool there isn't a mechanism to check the syntax

and the semantics of the *TatjaML document*. Since users of TATJA can have their own *custom made* TatjaML document and they can also add more definitions to the existing document, I think that is would be of a great assistant to the users of TATJA if the tool will have a function to check the syntax and semantics of this important definition document.

Currently a user can playback a test case that was recorded for a particular Applet on a different Applet. Although this operation doe not crash or do any harm to the system and to the Applets being tested, I think that TATJA should be able to recognise the Applet on which to run a particular test case. If there will be an improved version of TATJA, this issue should be tacked with priority.

A possible research that can lead to an improvement on the tool is in the way that TATJA uniquely identifies each component of the Applet to be tested. With the current system, if someone removes and adds some components to the Applet to be tested, it might be thee case that some of the components will be constrained to change their identification name, resulting in some of the previously recorded test cases to be irrelevant.

7.3 Conclusion

This project has given me the opportunity to learn more about software testing – a practice that is gaining more popularity as software clients are accepting more robust and reliable products from the software industry.

This project has also allowed me to improve my skills as regards to the object-oriented paradigm especially in the area of design. I also learnt new technologies such as the Reflection API, which I was totally unaware of, prior the commencement of this project.

The experience of research, designing, implementing, and documenting a project of this size has proved to be also an exercise in project and time management – a skill that for sure is required by each and every IT person that works in industry.

References

[1] Pressmen, R. S. & Ince I. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2000.

[2] Douglas Bell. *Software Engineering: A Programming Approach*. Addison-Wesley, 2000.

[3] Glenford J. Myers. *The Art of Software Testing (2nd Edition)*. John Wiley & Sons, 2004.

[4] Elfriede Dustin. *Effective Software Testing: 50 Specific Ways To Improve Your Testing*. Addison Wesley, 2002.

[5] WWW. Web Test Tools.
http://www.softwareqatest.com/qatweb1.html#OTHER

[6] WWW. Jakarta Cactus
http://jakarta.apache.org/cactus/

[7] WWW.
http://formflood.sourceforge.net/

[8] WWW. HtmlUnit.
http://htmlunit.sourceforge.net/

[9] WWW. HttpUnit Home.
http://www.httpunit.org/

[10] WWW. JMeter.
http://jakarta.apache.org/jmeter/

[11] WWW.
http://maxq.tigris.org/

[12] WWW. Web Application Load, Stress & Performance Testing.
http://www.loadtestingtool.com/

[13] WWW. Selenium.
http://selenium.openqa.org/

[14] WWW. iMacros.
http://www.iopus.com/imacros/web-testing.htm

[15] WWW iOpus.
http://www.iopus.com/

[16] WWW. Extensible Markup Language (XML).
http://www.w3.org/XML/

[17] WWW. W3C Math Home.
http://www.w3.org/Math/

[18] WWW. The GraphML File Format.
http://graphml.graphdrawing.org/

[19] WWW. MusicXML Definition.
http://www.recordare.com/xml.html

[20] WWW. Reflection (computer science).
http://en.wikipedia.org/wiki/Reflection_%28computer_science%29

[21] WWW. Trial: The Reflection API.
http://java.sun.com/docs/books/tutorial/reflect/

[22] WWW. OOPSLA 2003.
http://www.oopsla.org/oopsla2003/files/tut-27.html

[23] Deitel H. M., Deitel P. J. & Goldberg A. B. *Internet & World Wide Web: How To Program*. Prentice Hall, 2004.

[24] WWW. Car Loan Calculator.
http://mainline.brynmawr.edu/Courses/cs110/spring2002/Applets/CarLoan/CarLoan.html

[25] WWW. Damien Martin's physics website.
http://lifshitz.ucdavis.edu/~dmartin/index.html

[26] WWW. Slope Applet: First physics Applet.
http://lifshitz.ucdavis.edu/~dmartin/teach_java/slope/first_physics.html

[27] WWW. Robot (Java 2 Platform SE v.14.2).
http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Robot.html

[28] Harry M. Sneed. *The Rationale for Software Wrapping*. IEEE, 1997.

[29] Edward Miller. *WebSite Testing*. eValid, 2008.

[30] WWW. World Wide Web Consortium
http://www.w3c.org

[31] Alan M. Davis. *201 Principles of Software Development*. McGraw Hill, 1995.

[32] WWW. The W3C Markup Validation Service.
http://validator.w3.org/

[33] WWW. W3C Link Checker.
http://validator.w3.org/checklink

[34] WWW. The W3C CSS Validation Service.
http://jigsaw.w3.org/css-validator/

[35] WWW. Website Security – Acunetix Web Vulnerability Scanner.
http://www.acunetix.com/

[36] WWW. Application Security, Risk Assessment and Risk Mitigation
http://www.securityinnovation.com/

[37] WWW. Web Application Security and Testing from SPIy Dynamics
http://www.spydynamics.com/

[38] WWW. OWASP.
http://www.owasp.org/

[39] WWW. OWASP Top Ten Project
http://www.owasp.org/index.php/OWASP_Top_Ten_Project

Bibliography

John D. McGregor & David A. Sykes. *A Practical Guide To Testing Object-Oriented Software.* Addison Wesley, 2005.

Bill Hetzel. *The Complete Guide to Software Testing (2nd Edition).* John Wily & Sons, 1998.

Mauro Pizze & Michael Young. *Software Testing and Analyses: Process, Principles and Techniques.* John Wily & Sons, 2008.

Jeff Tian. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement.* John Wily & Sons, 2005.

Appendix A
User Manual

A.1 Requirements

TATJA is compatible with the Java Runtime Environment 1.6 (jre 1.6) or higher. In order to execute TATJA on a web browser it is required to grant all permissions to the *Applet Security Manager*. This can be achieved by replacing the contents of the *java policy file* that is found under the directory:

```
<JAVA_HOME>\lib\security\java.policy
```

...with the following contents:

```
grant {
     permission java.security.AllPermission;
};
```

A.2 Installation and Execution Instructions

TATJA consists of a JAR file, a resources folder and a simple HTML page. These are available on the CD distributed with this documentation and online on the TATJA home page and can be downloaded from http://tatja.matthewxuereb.com/.

TATJA is designed to work in a Java development environment, that is, in Integrated Development Environments (IDEs). In order to be able to use TATJA, the TATJA resources folder and the TATJA HTML page has

to be stored at the root of the Java Applet application project that has to be tested. The JAR should be placed in the same directory where the class files are stored. It is important that the JAR file is also added to the build path of the project.

In order to run TATJA the TATJA HTML file has to be executed in any web browser, such as Opera, Safari, Firefox, and more.

A.3 TATJA in Record Mode

A.3.1 The Record Toolbar

Figure 28 below illustrates a screen shot of the tool bar of TATJA in Record Mode. As it can be shown in Figure 28, nearly all of the buttons and icons are disabled except for the load Applet button and the help icon. Once an Applet is loaded inside the tool, the buttons and icons start to be enabled. This helps the user not to be confused which step to do next in the test case recording process.

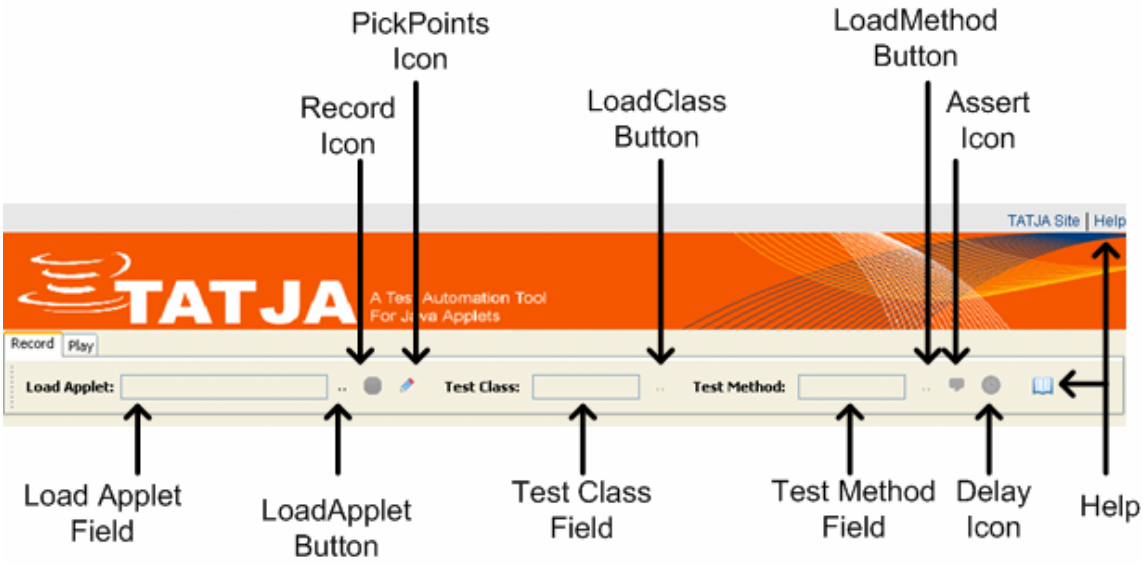

Figure 28: Screen shot of TATJA Record toolbar

The next section of this user manual makes reference to the above toolbar illustrated in Figure 28.

A.3.2 Loading the Applet to be tested

In order to be able to record test cases on a particular Applet, the Applet has to be first loaded in the testing tool. In order to load the Applet, the user has to click on the *LoadApplet Button*.

When the user clicks on this button, a dialog box as shown in Figure 29 is displayed on the screen. The user has to select a package from where to load the Applet. Once the user selects a particular package from the '*Load Applet From*' combo box, the '*Select Applet*' list box is populated with all the available Java Classes. The user has to select a particular class that is an Applet from the list box and clicks on ok. Once the user hits the '*Ok*' button, the dialog box is not displayed any more and the Applet to be tested is loaded within the tool.

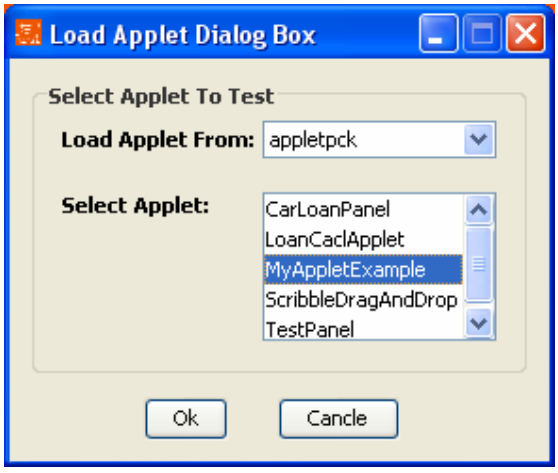

Figure 29: Screen shot of the Load Applet Dialog Box

A.3.2 Loading/Creating TATJA Test Classes

Once the user has loaded the Applet to be tested in TATJA, the next step is that of loading or creating a new test class where to store the recorded test cases. In order to do this the user has to click on the '*LoadClass Button*'. When this button is clicked, a dialog box as shown in Figure 30 is displayed on the screen. From this dialog box the user can selects and makes use of an existent test class. Else, the user can create a new test class.

To use an existent test class, the user has to first select the package from where to load the test class. The package is selected from the *'Select package'* combo box. Then the user has to select the class from the *'Choose class'* combo box.

In order to create a new class the user has to first either select the package where to store the new class by making use of the *'Select Package'* combo box or else creates a new package by typing the name of the new package in the *'New Package'* text field. Then the user has to type in the name of the new test class in the *'New Class'* text field.

In order to continue with the testing process, the user has to click on the *'Ok'* button. Once the user clicks in this button, the dialog box is removed and the user can proceed to the next step of the recording process.

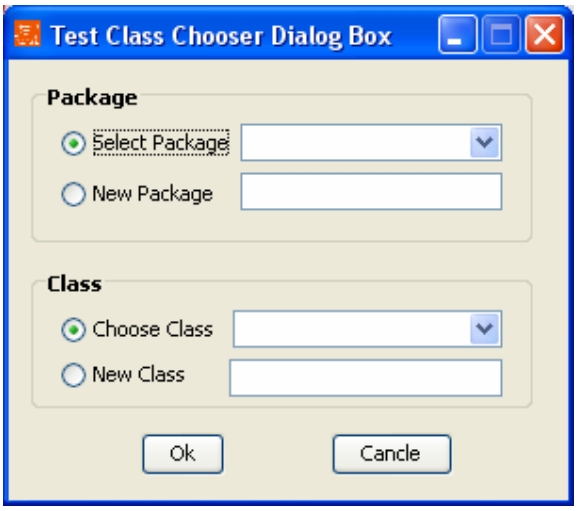

Figure 30: Screen shot of the Test Class Chooser Dialog Box

A.3.3 Selecting/Creating TATJA Test Methods

Once the user has created or selected the test class, the next step is to select or create new test methods where to record the test cases. In order to do this, the user has to click in the *'TestMethod Button'*. When the user clicks on this button, a dialog box as illustrated in Figure 31 is displayed on the screen. From this dialog box the user can either create a new test method or else use an existent method. In order to select an existing method (if there are any available methods) the user has select a method from the list box on the left hand side of the dialog box and then clicks on the *'Ok'* button. In order to create a new test method, the user has to type in the name of the new method and an optional description on the text fields that are available of the right hand side of the dialog box. When the user clicks on the *'Ok Button'*, the dialog box is removed and the user can proceed. Note that the user can also select an existing test method by typing the name of the test method in the *'Test Method Text Field'* in the Record Toolbar.

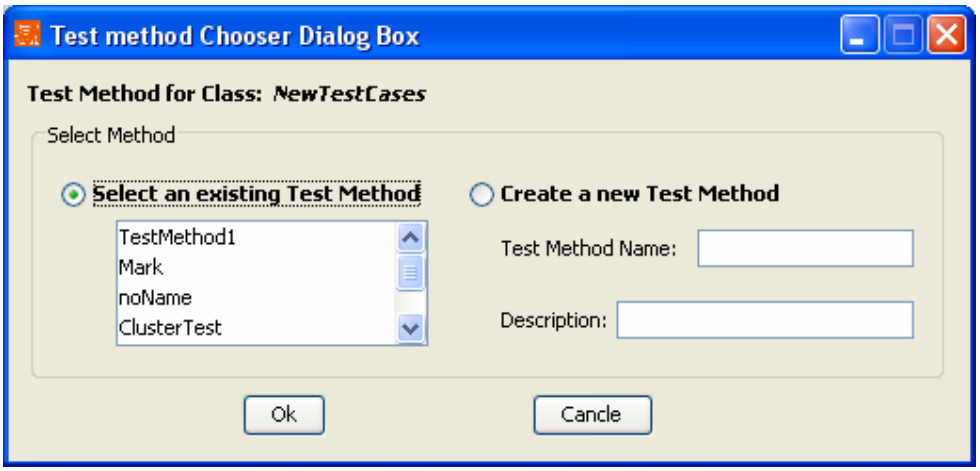

Figure 31: Screen shot of the Test Method Chooser Dialog Box

A.3.4 Test Case Recording

Once the Applet to be tested is loaded and the test class and test method are selected, the user can star to record the test cases. Note that from this stage onwards, the user can view the contents of the test method or test class by simply placing the mouse pointer on their respective text field on the record toolbar. When the mouse pointer is placed on these fields the contents of the test class or test method are displayed on a popup window. Note that once the user removes the mouse pointer from this window, the window is removed. In order to keep this window even when the mouse pointer is moved to a different position, the user has to press the keyboard *F2* key. Figure 32 below shows a screen shot of such window.

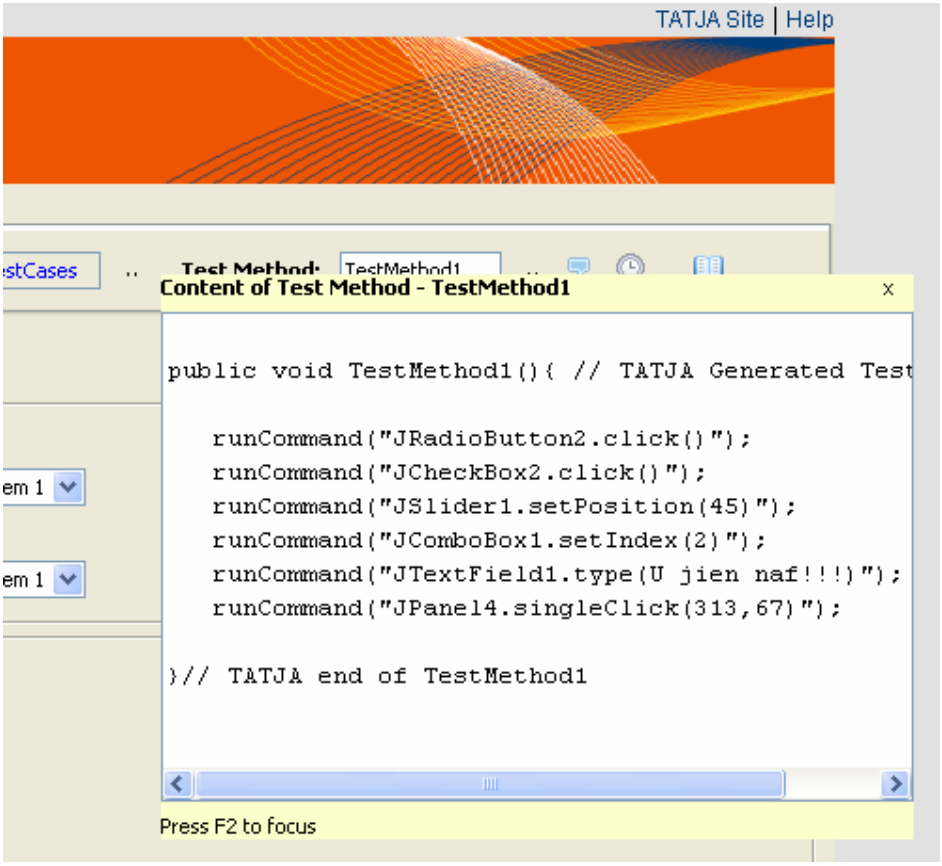

Figure 32: A screen shot of the window with the contents of a particular Test Method as displayed when the mouse pointer is displayed on the *Test Method* text field

In order to record a test action, the user has to click on the '*Record Icon'*. When the user clicks on the record icon, TATJA switches into recording mode. When TATJA is in recording mode and the user moves the mouse pointer on the Applet to be tested, the component underneath the mouse pointer highlights itself and a popup window is displayed on the screen with all the possible actions that the tester can perform on it.

Figure 33 below shows the window that is displayed when the user places the mouse pointer on a *JSliderBar* component. As it is illustrated in the screen shot, the component is highlighted in pink and the *Recorder Tool Panel* window is displayed on the screen. At the top part of the window the details of the component are displayed. At the bottom of the component the possible actions together with their

respective descriptions are displayed. In order to record a particular action, the user has to click on one of these actions.

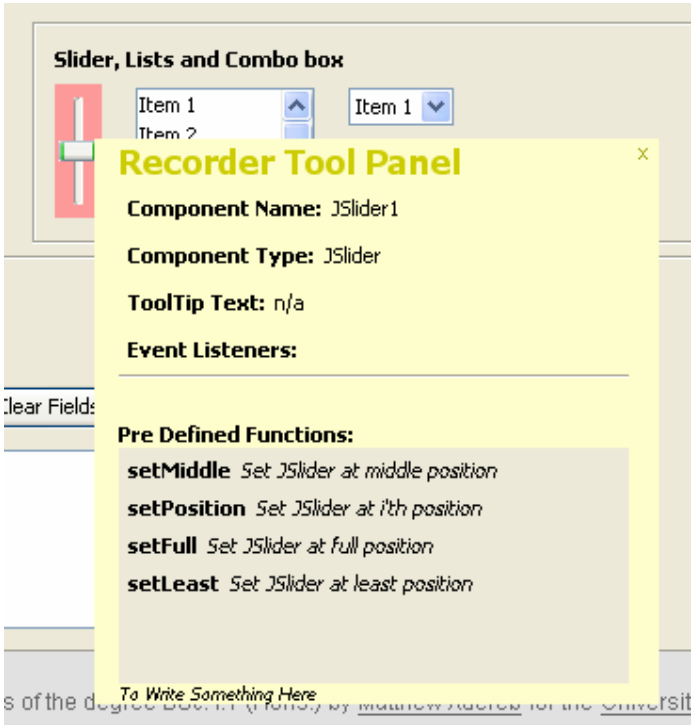

Figure 33

When the user clicks on a particular action, a new dialog box is displayed on the screen. Figure 34 below shows a screen shot of the dialog that is displayed when the user selects the *setPosition* action from the above example of Figure 33.

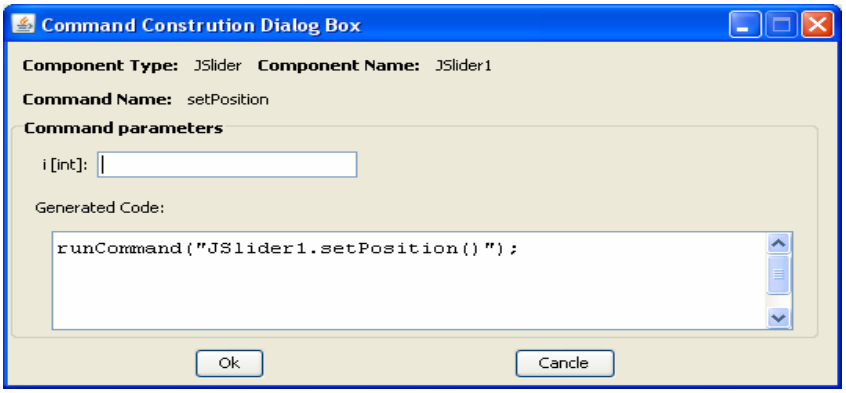

Figure 34

The purpose of the dialog box illustrated in Figure 34 is in order to let the user to input any required parameters. The required input parameters are inputted from the text fields generated 'tail made' for the specific action command at the middle part of the dialog box. At the bottom part of the dialog box the actual generated Java code is displayed in a text area. The user can add any valid java code in the text area in order to further enhance the recorded test cases. When the user clicks on the *'Ok'* button, the generated Java code is appended to the test method.

Sometimes it is required that between an action and another there is a delay. In order to include a delay in the test cases, the user has to click on the *'Delay Icon'* from the Record Toolbar. When the user clicks on this icon, a dialog box as shown in Figure 35 below is displayed on the screen. The user has to type in the number of seconds to delay and then clicks on the *'Ok'* button.

Figure 35

With TATJA it is also possible to include *Assert Statements* within the recorded test cases. In order to include an assert statement, the user has to click on the *'Assert Icon'* from the Record Toolbar. A dialog box as shown in Figure 36 is displayed on the screen. In order to complete an assert statement, the user has to enter the required information and clicks on the *'Ok'* button.

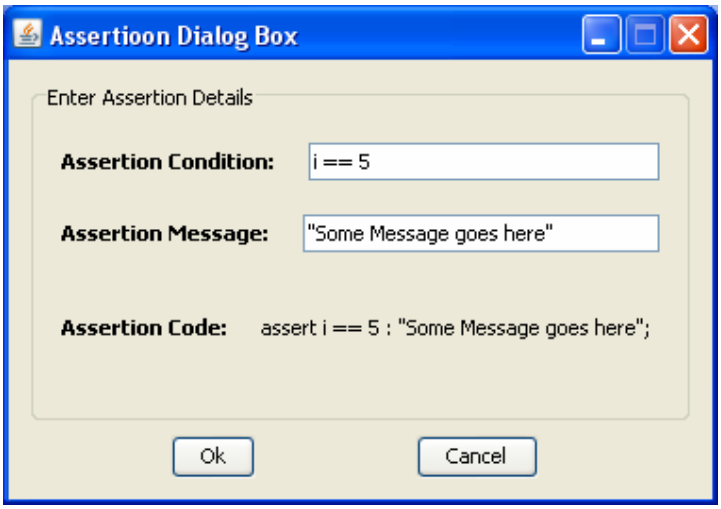

Figure 36

A.3 TATJA in Playback Mode

A.3.1 The Playback Toolbar

Figure 280 below illustrates a screen shot of the toolbar of TATJA in Playback Mode. This section of this user manual makes reference to the toolbar illustrated in Figure 37.

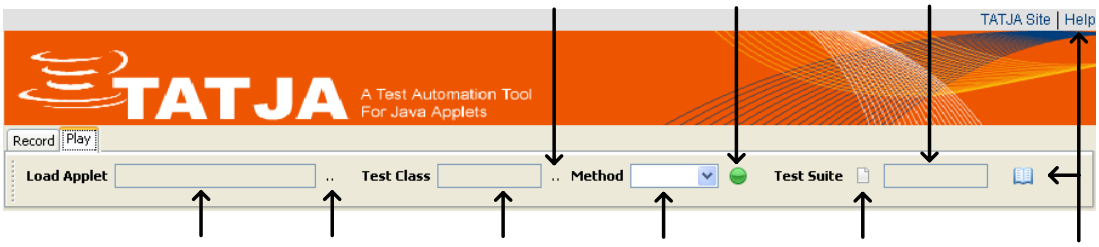

Figure 37

A.3.2 Loading Applet and Test Class

The Applet to be tested and the test class are loaded in the same way as they are loaded during the Recording operation. The *'LoadApplet Button'* is clicked in order to open the *Load Applet Dialog Box* while the *'LoadClass Button'* is clicked in order to open the *Load Test Class Dialog Box*.

A.3.2 Executing a Test Case

In TATJA a test case is equivalent to a Java method. In order to execute a particular test case, first the Applet to be tested and the test class has to be loaded. Once these are loaded, the test case (method) to be executed is selected through the *'SelectMethod Combo box'*.

To playback, the selected test method, the green *'Playback Icon'* is pressed. When this icon is pressed the Java method is executed and thus the recorded test case is play backed.

A.4 TATJA Test Suite

With TATJA one can create test suites. A test suite is a collection of test cases that can be play backed programmatically without the need of manually loading TATJA and the Applet to be tested. By using TATJA Test Suits, the user can programmatically load the Applet to be tested encapsulated within a micro version of TATJA on any web browser and playback the recorded test cases.

A.4.1 Creating a Test Suite

When creating a Test Suite, a new Applet wrapper class and an HTML page that executes this class are created by TATJA. To create a Test

Suite, the *Play Tab* of TATJA is first selected. Then the Applet to be tested is loaded in the tool. In order to create the Test Suite, the *'TestSuite Icon'* is clicked. When this icon is clicked, the *New Test Suite Dialog Box* is displayed on the screen as shown in Figure 38 below.

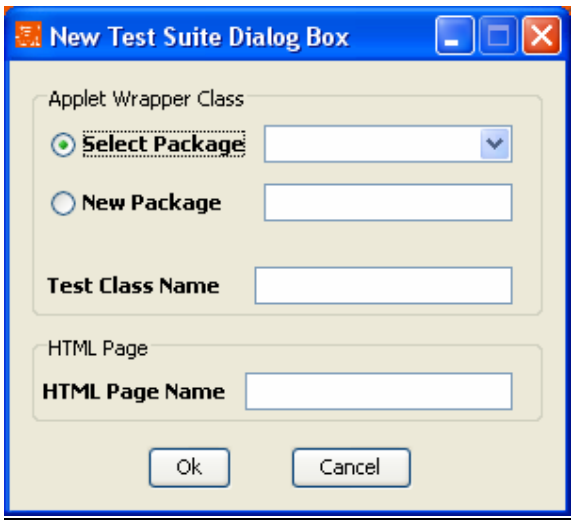

Figure 38

The name of the package where to store the test suite class is selected from the *Select Package* combo box. Otherwise the user can creates a new package by typing in the name of the new package where to store the new test suite class in the *New Package* text field. The name of the test suite class is entered in the *Test Class Name* field and the name of the HTML page that executes the Applet is entered in the *HTML Page Name* text field.

A.4.2 Editing/Programming a Test Suite

When all the required information is entered, a new Java class and a new HTML document are created. The Java class is stored in the package selected previously while the HTML document is stored at the root of the project's directory. The generated Java class consists of a

constructor and an empty method named *tatjaSuiteTestMethod()* very similar to the code snippet shown below.

```
public class SlopeTestSuite extends JApplet{

	private RecorderHandler recorderHandler;
	private Class c;
	private JApplet test;
	private Map<String,Component> componentMapTable;

	public SlopeTestSuite(){
		// Setting the windows look and feel
		try {
		    UIManager.setLookAndFeel
		    ("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
		}catch(Exception e) {
			e.printStackTrace();
		}

		try{
			c = Class.forName("sloap.SloapApplet");
			test = (JApplet)c.newInstance();
			this.add(test);
			recorderHandler = new RecorderHandler(test);
			recorderHandler.disableRecorderMouseListener();
			componentMapTable =
					recorderHandler.getComponentsMapTable();
		}catch(Exception e){
			JOptionPane.showMessageDialog(this,e.getMessage(),
			    "Error Message",JOptionPane.ERROR_MESSAGE);
		}
	}

   public void tatjaSuiteTestMethod(){ // TATJA Generated tatjaSuiteTestMethod

   } // TATJA end of tatjaSuiteTestMethod
}
```

The calls to the recorded test cases should be added in the empty *tatjaSuiteTestMethod* as shown in the following code snippet example:

```
public void tatjaSuiteTestMethod(){ // TATJA Generated tatjaSuiteTestMethod
   SlopeTest s = new SlopeTest(test,componentMapTable);
   s.TestMethod1();
} // TATJA end of tatjaSuiteTestMethod
```

A.4.3 Executing a Test Suite

When the *tatjaSuiteTestMethod* is ready from the required addendums and compiled, the test suite can be executed by loading the generated

HTML page in any web browser. This can be done programmatically by using the *TATJAConsole* API through the method:

```
public static void runFileInWebBrowser(String browserConstant,String filePath)
```

The file path of the HTML document should be passed as parameter through argument *filePath*. The browser to be used to run the test should be identified by passing the string constant given in the *WebBrowserConfigs.properties* as parameter through the argument *browserConstant*. In the *WebBrowserConfigs.properties* the shell commands that invokes and loads the browsers (Operating System specific) are stored.

Appendix B

TatjaML Sample Document

```
<?xml version="1.0" ?>

<TatjaML>
	<component name="JLabel">
		<commands>
		</commands>
		<events>
			<singleClick params="int,int" desc="Single Click on a JLabel">
				<listener>MouseListener</listener>
				<invoke method="mouseClicked">
					<pass_object object="MouseEvent">
						<param type="java.awt.Component">^</param>
						<param type="int">500</param>
						<param type="long">0</param>
						<param type="int">16</param>
						<param type="int">&x</param>
						<param type="int">&y</param>
						<param type="int">1</param>
						<param type="boolean">true</param>
					</pass_object>
				</invoke>
			</singleClick>
		</events>
	</component>

	<component name="JPanel">
		<commands>
		</commands>
		<events>
			<singleClick params="int,int" desc="Single Click on a JPanel">
				<listener>MouseListener</listener>
				<invoke method="mouseClicked">
					<pass_object object="MouseEvent">
						<param type="java.awt.Component">^</param>
						<param type="int">500</param>
						<param type="long">0</param>
						<param type="int">16</param>
						<param type="int">&x</param>
						<param type="int">&y</param>
						<param type="int">1</param>
						<param type="boolean">true</param>
					</pass_object>
				</invoke>
			</singleClick>
		</events>
	</component>

	<component name="JTextField">
		<commands>
			<type params="String" desc="Fill JTextField">
```

```
                        <return>null</return>
                        <invoke method="setText">
                                <param type="java.lang.String">&text</param>
                        </invoke>
                </type>

                <getText params="" desc="Get contents of JTextField">
                        <return>String</return>
                        <invoke method="getText">
                        </invoke>
                </getText>
        </commands>
        <events>
        </events>
</component>

<component name="JTextArea">
        <commands>
                <type params="String" desc="Fill JTextArea">
                        <return>null</return>
                        <invoke method="setText">
                                <param type="java.lang.String">&text</param>
                        </invoke>
                </type>

                <getText params="" desc="Get contents of JTextArea">
                        <return>String</return>
                        <invoke method="getText">
                        </invoke>
                </getText>
        </commands>
        <events>
        </events>
</component>

<component name="JCheckBox">
        <commands>
                <check params="boolean" desc="Set JCheckBox selected">
                        <return>null</return>
                        <invoke method="setSelected">
                                <param type="boolean">true</param>
                        </invoke>
                </check>

                <unCheck params="boolean" desc="Set JCheckBox not selected">
                        <return>null</return>
                        <invoke method="setSelected">
                                <param type="boolean">false</param>
                        </invoke>
                </unCheck>

                <click params="" desc="Click on JCheckBox">
                        <return>null</return>
                        <invoke method="doClick">
                        </invoke>
                </click>

        </commands>
        <events>
        </events>
</component>

<component name="JRadioButton">
        <commands>
                <check params="boolean" desc="Set JRadioButton selected">
                        <return>null</return>
                        <invoke method="setSelected">
```

```
                              <param type="boolean">true</param>
                        </invoke>
                  </check>

                  <unCheck params="boolean" desc="Set JRadioButton not selected">
                        <return>null</return>
                        <invoke method="setSelected">
                              <param type="boolean">false</param>
                        </invoke>
                  </unCheck>

                  <click params="" desc="Click on JRadioButton">
                        <return>null</return>
                        <invoke method="doClick">
                        </invoke>
                  </click>
            </commands>

            <events>
            </events>
      </component>

      <component name="JButton">
            <commands>
                  <getName params="" desc="Get the name of the JButton">
                        <return>String</return>
                        <invoke method="getText">
                        </invoke>
                  </getName>

                  <isClickable params="" desc="Check if the JButton is enabled">
                        <return>boolean</return>
                        <invoke method="isEnabled">
                        </invoke>
                  </isClickable>
            </commands>

            <events>
                  <click params="" desc="Generate a click event on a JButton">
                        <listener>ActionListener</listener>
                        <invoke method="actionPerformed">
                              <pass_object object="ActionEvent">
                                    <param type="java.lang.Object">^</param>
                                    <param type="int">3</param>
                                    <param
type="java.lang.String">^getActionCommand</param>
                              </pass_object>
                        </invoke>
                  </click>
            </events>
      </component>

      <component name="JToggleButton">
            <commands>
                  <getName params="" desc="Get the name of the JToggleButton">
                        <return>String</return>
                        <invoke method="getText">
                        </invoke>
                  </getName>
            </commands>

            <events>
                  <click params="" desc="Generate a click event on a
JToggleButton">
                        <listener>ActionListener</listener>
                        <invoke method="actionPerformed">
                              <pass_object object="ActionEvent">
```

```
                                        <param type="java.lang.Object">^</param>
                                        <param type="int">3</param>
                                        <param
type="java.lang.String">^getActionCommand</param>
                                    </pass_object>
                            </invoke>
                        </click>
                </events>
        </component>

        <component name="JSlider">
                <commands>
                        <setMiddle params="int" desc="Set JSlider at middle position">
                                <return>null</return>
                                <invoke method="setValue">
                                        <param type="int">50</param>
                                </invoke>
                        </setMiddle>

                        <setLeast params="int" desc="Set JSlider at least position">
                                <return>null</return>
                                <invoke method="setValue">
                                        <param type="int">0</param>
                                </invoke>
                        </setLeast>

                        <setFull params="int" desc="Set JSlider at full position">
                                <return>null</return>
                                <invoke method="setValue">
                                        <param type="int">100</param>
                                </invoke>
                        </setFull>

                        <setPosition params="int" desc="Set JSlider at i'th position">
                                <return>null</return>
                                <invoke method="setValue">
                                        <param type="int">&i</param>
                                </invoke>
                        </setPosition>
                </commands>
                <events>
                </events>
        </component>

        <component name="JComboBox">
                <commands>
                        <setIndex params="int" desc="Select JComboBox item at i'th
index">
                                <return>null</return>
                                <invoke method="setSelectedIndex">
                                        <param type="int">&i</param>
                                </invoke>
                        </setIndex>

                        <setItem params="java.lang.Object" desc="Select JComboBox s
value">
                                <return>null</return>
                                <invoke method="setSelectedItem">
                                        <param type="java.lang.Object">&s</param>
                                </invoke>
                        </setItem>
                </commands>

                <events>
                </events>
        </component>
```

```
	<component name="JList">
		<commands>
			<setIndex params="int" desc="Select i'th JList index">
				<return>null</return>
				<invoke method="setSelectedIndex">
					<param type="int">&i</param>
				</invoke>
			</setIndex>

			<setIndices params="int[]" desc="Select JIten indices">
				<return>null</return>
				<invoke method="setSelectedIndices">
					<param type="int[]">&i</param>
				</invoke>
			</setIndices>

			<setItem params="java.lang.Object,boolean" desc="Select JItem
value">
				<return>null</return>
				<invoke method="setSelectedValue">
					<param type="java.lang.Object">&i</param>
					<param type="boolean">true</param>
				</invoke>
			</setItem>
		</commands>

		<events>
		</events>
	</component>
</TatjaML>
```

Appendix C

Contents of the CD ROM

\	
\student.txt	My Details
\TATJA	The TATJA project folder and TATJA.html
\TATJA\src	The Source Code of the project
\TATJA\jar	The jar file of TATJA
\TATJA\bin	The compiled classes of TATJA
\TATJA\recourses	The resources folder of TATJA
\TATJA\documentation	The documentation

Appendix D

Permissions and Clarifications

D.1 Permission to use working Applets

In Chapter 6 – *Testing and Evaluation*, three working Java Applets are used as test cases. The first Applet that is used, the *Clustering Simulation Applet*, was developed by the author of this dissertation as an assignment. The other two Applets, the *Car Loan Applet* and the *Frictionless Slope Simulation Applet* are can be downloaded from [24] and [25] respectively. Permissions from their respective authors were given to the author of this dissertation by email. Copies of the received emails are included below:

D.1.1 Permission email form Mr. Damien Martin

```
Dear Matthew,

  I am pleased that my slope applet is useful to someone out there. Please
feel free to use it as part of your BSc (Hons) dissertation.

  Best regards,

Damien Martin

> Dear Mr. Martin,
>
> I am a 4th Year student at "The University of Malta
> (http://www.um.edu.mt)" reading a BSc IT (Hons.) degree.  I am currently
> working on my dissertation entitled "A Test Automation Tool For Java
> Applets".  The tool is ready and currently I am in the process of testing
> it on working Applets.
>
> While I was browsing through the internet I came across your 'Slope
> Applet'
> which is a very good example of a working Applet.  Can you kindly give me
> permission to use you Applet for my dissertation demonstration?  I will
```

```
> obviously reference it as appropriate.
>
> Regards
> Matthew Xuereb
>
> P.S. Please note that my Applet testing tool is not a commercial tool and
> if you give me permission to use your Applet it is going to be used
purely
> for academic purposes.
>
```

D.1.2 Permission email form Profs. Deepak Kumar

```
Hi Matthew:

Feel free to use it as you wish. And, good luck with your dissertation.

Best,

Deepak.

MATTHEW XUEREB wrote:
> Dear Mr Kumar,
>
> I am a 4th Year student at "The University of Malta
> (http://www.um.edu.mt)" reading a BSc IT (Hons.) degree.  I am currently
> working on my dissertation entitled "A Test Automation Tool For Java
> Applets".  The tool is ready and currently I am in the process of testing
> it on working Applets.
>
> While I was browsing through the internet I came across your 'Car Loan
> Applet'
> which is a very good example of a working Applet.  Can you kindly give me
> permission to use you Applet for my dissertation demonstration?  I will
> obviously reference it as appropriate.
>
> Regards
> Matthew Xuereb
>
> P.S. Please note that my Applet testing tool is not a commercial tool and
> if you give me permission to use your Applet it is going to be used
purely
> for academic purposes.
>
>


--
Deepak Kumar
Professor of Computer Science
Bryn Mawr College, Bryn Mawr, PA 19010
dkumar@cs.brynmawr.edu, dkumar@acm.org
http://www.cs.brynmawr.edu/~dkumar
```

D.2 The TATJA Logo

The author of this dissertation would like to clarify that the logo of this tool is inspired by the Java logo. However this logo is not copied directly from the Java logo.